

Vantage Sequence Programming Tutorial

Ron Daigle - July, 2017

Updated for Vantage Software Release 3.x

Table Of Contents

1.0 Introduction	3
2.0 System Components	4
3.0 Signal Path Components	6
4.0 A First Sequence Program	10
4.2 The Transmit Waveform (TW) definition	15
4.3 The TX object definition	16
4.4 The TGC object definition	17
4.5 The Receive object definition	18
4.6 The Event object definition	20
5.0 Running the Program	22
6.0 Acquiring multiple acquisition data with the Vantage system	31
7.0 Using an External Function	38
8.0 Running with Acquisition Hardware	42
8.1 Operation of the Hardware and Software Sequencers	42
8.2 Running the example script with the hardware.	45
9.0 Running a Sequence Continuously	48
10.0 Adding a GUI Control	52
11.0 Designing an Asynchronous Script	58
12.0 Performing Image Reconstruction	65
12.1 Defining the coordinate system with respect to the probe.	65
12.2 The PData object definition	66
12.4 The Recon and ReconInfo object definitions	72
13.0 Processing the Reconstructed Data.	76
14.0 The Complete Acquisition and Image Processing Script	82

1.0 Introduction

The Verasonics Vantage Research Systems are designed to be a flexible tool for transmitting, receiving and processing ultrasound information. Essentially all aspects of a modern commercial ultrasound system are able to be controlled by the user, and it is possible to create new and novel methods of ultrasound acquisition and processing. The new user may find the complexity of the system somewhat daunting, especially if one's knowledge of how ultrasound systems work is somewhat limited. If this is the case, it is recommended that the user first study some of the many reference works¹ in this area before attempting a sequence program.

This tutorial will attempt to gradually introduce users to the concepts and programming techniques that are needed to generate useful sequences for acquiring and processing ultrasound data. It is meant to complement the Vantage Sequence Programming Manual, which has lots of detailed information, but is more suited for reference than learning how to use the system. The basic programming concepts will be covered here, and with this understanding, it should be easier to digest the material in the Sequence Programming Manual and example scripts.

Our focus in this tutorial will be on writing the programs that define a sequence of events for execution by the Verasonics Vantage Research System, and its associated host computer. The Vantage unit consists of all of the hardware in the Vantage Hardware Module, which is connected to the host computer by the PCI express cable. The scripts that we will write to program the system are written in the Matlab™ language, which provides a nice environment for defining the various programming objects contained in a sequence of events. Our scripts will define the order of actions to be carried out in the Vantage unit, as well as actions that will take place in the software environment of the host computer. It is assumed that the user has a familiarity with programming using the Matlab language and the Matlab programming environment.

¹ An excellent ultrasound reference book is "Diagnostic Ultrasound Imaging: Inside Out," 2nd Edition by Thomas L. Szabo, Elsevier Academic Press.

2.0 System Components

Before beginning the programming of event sequences, it is necessary to become familiar with the components of the system. Within the Vantage Unit, there are several hardware components that will be referred to in later discussions. These components are shown in Fig. 2.1 below. Note that there are no user serviceable components in the Vantage unit, and disassembling the system may void the system warranty.

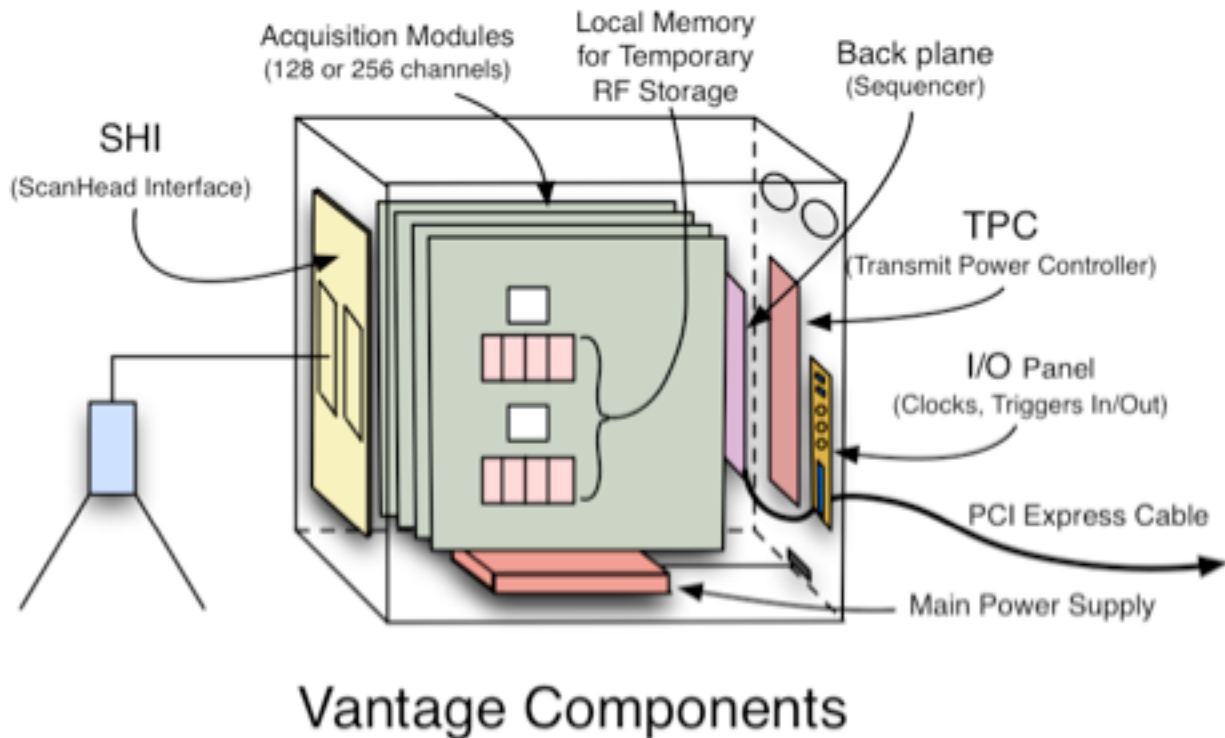


Fig. 2.1 Vantage Unit hardware components (256 channel system shown).

Scanhead Interface (SHI) - This is the module that contains the transducer connector(s). There is a single connector for a 64 or 128 channel Vantage system and dual connectors for a 256 channel system. Starting in late 2015, the transducer connectors are mounted on a user removal module called the UTA (Universal Transducer Adapter), which can be changed to support different connector types. The UTA/SHI module also contains the circuitry for detecting the presence of a scanhead plugged into a connector, and for programming any high voltage multiplexers contained in the scanhead body or connector.

Acquisition Modules - These are the main electronics boards for the Vantage unit, and contain the circuitry for transmitting and receiving ultrasound signals for multiple channels. Each module supports up to 64 transmitters and 64 receive channels. The received ultrasound signals are digitized and stored in **local memory** on the Acquisition Modules prior to transfer to the host computer. The Acquisition Modules also provide the per channel digital filtering and signal conditioning that can be applied after A/D conversion.

Back Plane Module - This module contains the **hardware sequencer** that controls the operation of the Acquisition Modules. The module also distributes the PCI Express bus to the Acquisition Modules via a PCI express switch. The switch connects to the PCI Express Cable Connector on the I/O Panel that is used to connect to the host computer.

Transmit Power Controller (TPC) - The TPC module provides the high voltage power supply for the transmitters, as well as several other power supply levels used by the system.

I/O Panel - The I/O Panel provides the PCI express cable connection for connecting to the host computer. It also provides the BNC connections for two input triggers and one output trigger, as well as clock outputs for synchronization with external devices.

3.0 Signal Path Components

The sequence programmer must also be familiar with the signal path components of the system, since the attributes of these components must be specified for the various sequence events. Figures 3.1 to 3.3 below show the signal path components for various Vantage systems that reside on an Acquisition Module for a single receive channel and its associated transmitter(s). Each receiver writes its acquisition data to local memory on the Acquisition Module, and these data can then be moved to the host computer using a DMA (Direct Memory Access) transfer.

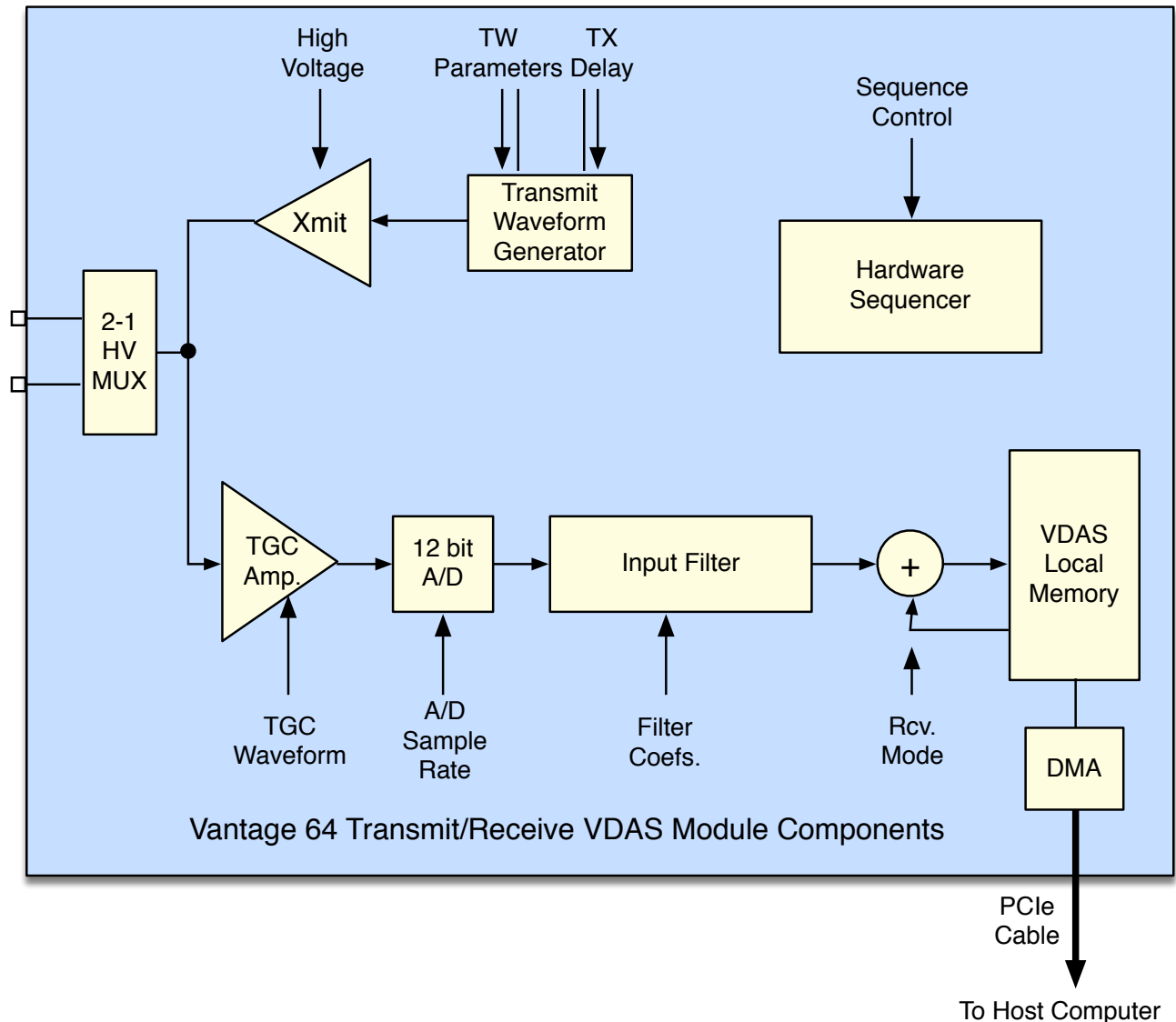


Fig. 3.1 Vantage 64 Signal Path Components

A Vantage 64 system has 64 transmitters and 64 receive channels with each transmit/receive pair connected by a high voltage multiplexor switch to one of two I/O channels of the 128 I/O connector. The I/O channels are 64 apart; eg. system transmit/receive channel one can be connected to either I/O channel one or I/O channel 65. The independent channel MUX programming allows the 64 transmit/receive system channels to be moved

across the 128 elements of a transducer array by one element at a time. The rotation of the channels can be abstracted from the user so that the user only needs to deal with a logical description of the aperture.

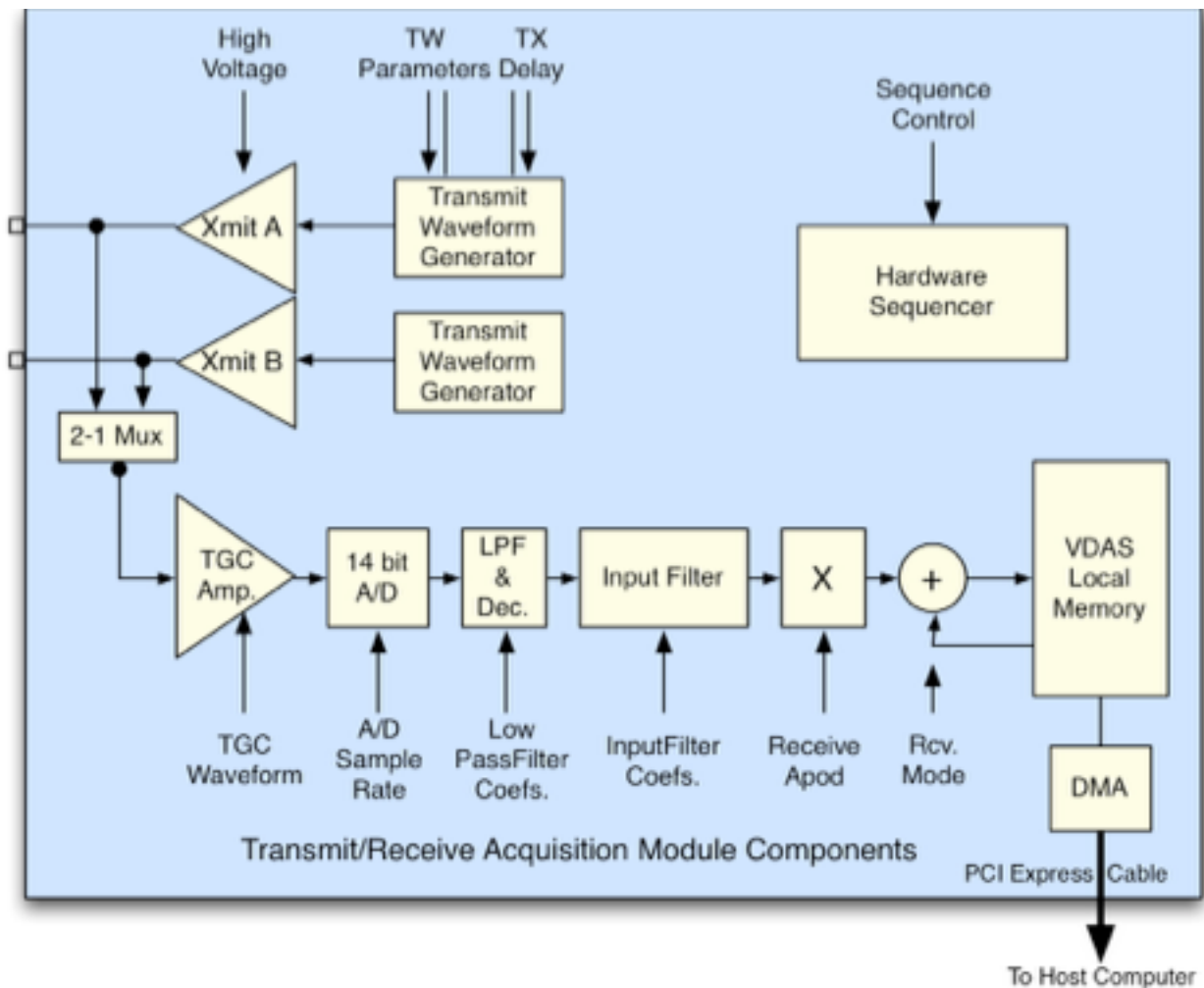


Fig. 3.2 Vantage 64LE Signal Path Components

A Vantage 64LE system has 128 transmitters that are connected to the 128 I/O channels of the single transducer connector. Each receive channel can connect to one of two I/O channels that are separated by 64 I/O channels. For example, receive channel one can connect to I/O channel 1 or I/O channel 65. For any transmit/receive acquisition, the receive channels must be connected to one or the other of their associated I/O channels, but by setting the individual connections for each receive channel, a 64 element aperture can be placed anywhere along a 128 element array. This configuration is shown in Fig. 3.2.

For a Vantage 128 Unit, the transmitters and receivers from the two Acquisition modules are connected in a one-to-one manner to the 128 I/O connections of the single transducer connector on the SHI module. For a 256 channel Vantage system, transmit and receive channels 1 to 128 of the Acquisition Modules are connected to the first connector's 128 I/O channels, while transmit and receive channels 129 to 256 of the Acquisition Modules are connected to the second connector's 128 I/O channels. Either transducer connector can

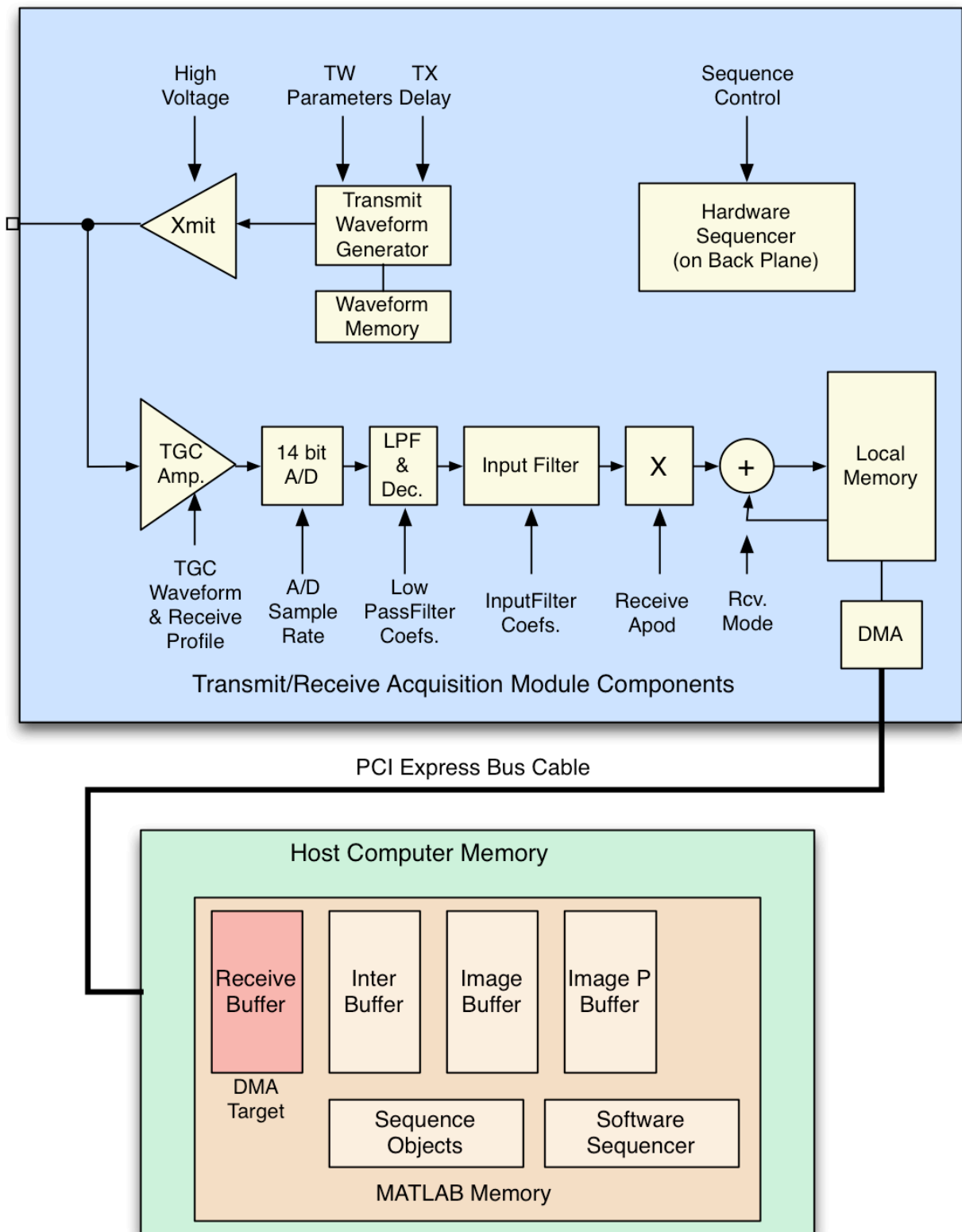


Fig. 3.3 Vantage 128 Signal Path Components, along with Host Software Objects.

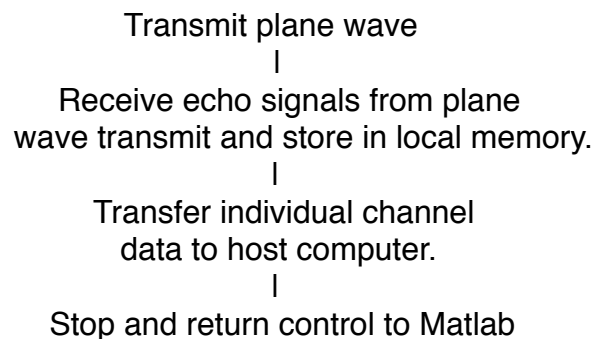
be used independently or they can both be used together, as when connecting a 256 element array. As we will see when we get into the sequence programming, these connections of transmit and receive channels to connector I/O channels will take on a logical definition, which will make it easier for the programmer to configure the system for various transducer types.

A typical acquisition event will consist of transmitting on one or more transducer elements, and receiving from some number of transducer elements. The received ultrasound signals are amplified as a function of time from the transmit event by the Time Gain Controlled Amplifiers, then digitized by the 14 bit analog-to-digital converters at an appropriate sample rate. The digitized signals are then processed by a block of digital filters, whose function is to shape the spectrum of the received signals and possibly reduce the bandwidth and sample rate. The conditioned signals are then scaled according to a receive apodization value and written to local memory on the Acquisition modules for temporary storage. Data previously written to local memory can be added to the incoming data if desired, allowing accumulation of multiple acquisitions. The storing of acquisition data to local memory occurs in real-time as the ultrasound signals are received, allowing for sequences of acquisition that occur at very high repetition rates for limited periods of time. At a user determined time in a sequence, the acquired data in local memory can be DMA'd to the host computer over the PCI express bus. The transfer of data to the host computer can occur concurrently with the acquisition of new data.

Also shown in Fig. 3.3 are the possible memory buffers that can be created in the Matlab memory space of the host computer. The Receive Buffer is shown in a different color to indicate that only it can be the target for the DMA transfers of acquisition data from the local memory on the Acquisition Modules. Once the acquisition data are transferred into the Receive Buffer, all processing actions consist of memory-to-memory processes, where data are accessed from one buffer, processed, and written to another buffer. For example, when sufficient RF data have been acquired to form an image, an image reconstruction process can process data from a Receive Buffer to an Inter or Image Buffer. If further image processing is required, an image processing process can process data from the Inter or Image Buffer to another Image Buffer or an Image P buffer. (The Image P Buffer is typically the destination for processed image data destined for display.) Finally, since the various buffers are all allocated within the Matlab memory space, it is possible for the user to access each buffer from within Matlab, for custom processing and to generate appropriate displays of data.

4.0 A First Sequence Program

We will now proceed to write a first sequence program. It is assumed here that the user is familiar with the Matlab programming language, and has installed the latest Verasonics software and Matlab on their host computer, which is connected to the Vantage Unit via the PCI express cable. Before writing the code for our sequence, it is a good idea to sketch out the various actions that we want our sequence to take. For this example, we will keep it simple with only a few actions. This program will transmit an unfocused burst of ultrasound, emitting a plane wave that propagates over the field of view of the transducer. We will acquire the RF receive data for all receive channels from the transmit event, and transfer these data to the host computer. Finally, we will stop our sequence and return control to Matlab.



Programming the Vantage Research System basically consists of defining various system parameters and attributes, and the sequence of events to be carried out by the hardware and software. These parameters are defined using Matlab™ structures in a setup script written in the Matlab programming language. The structures are then saved to a .mat file, where they will be accessed at the time that the program is run. To run the program, the VSX (Verasonics Script eXecution) program is run, and the name of the .mat file provided when asked for. VSX then parses the structures from the file, adding any missing attributes needed to program the system, then loads the sequence into the software and hardware sequencers, and tells the sequencers to run.

The new user can follow the development of this example in their Matlab environment. Open Matlab and make sure the main path for the Matlab command line is set to the current Verasonics software directory. Then execute the command 'activate' to set the proper Matlab paths for the directory. To create our sequence program, we first create a new .m file in the Matlab editor. This file can be created in the Vantage directory or another location that is added to the Matlab path, such as the ExampleScripts directory. It is generally wise to name the program after it's function, so we will name our file 'SetUpL11_4vAcquireRF.m'. We include the transducer that will be used in the name, so that we know which transducer to use when we run the program. Don't worry if you don't have an L11-4v transducer, as we can run our script without having the transducer present.

It is good to get in the habit of defining parameters and structures in a certain order, as the order is important for the correct parsing of the parameters by the software. It is also

necessary to start our scripts with a 'clear all' statement, as we will be saving all the structures created at the end of our script, and we don't want to save structures or variables left over from previous programs. We will start by defining some system parameters.

```
clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128;      % number of transmit channels.
Resource.Parameters.numRcvChannels = 128;   % number of receive channels.
Resource.Parameters.connector = 1;          % transducer connector to use.
Resource.Parameters.speedOfSound = 1540;    % speed of sound in m/sec
```

Our first statements specify the transmit and receive channels available for our system. In the example text above, we are setting the parameters for a 128 channel Vantage Unit, which has 128 transmitters and 128 receive channels. A 64 channel Vantage Unit would specify `Resource.Parameters.numTransmit = 64` and `Resource.Parameters.numRcvChannels = 64`, instead of 128. For a Vantage 64LE, only `Resource.Parameters.numRcvChannels` would be set to 64. When we load our program with VSX, it will verify that we have the resources specified, and abort with an error if the specified resources don't match the hardware (this resource checking is skipped when running in simulate mode). If the Vantage Unit is a 256 channel system, the line in red can be added to specify which scanhead connector will be used (defaults to 1 if not provided). Note that if we are using only one connector of a 256 channel system, the resources for that connector are 128 transmitters and 128 receivers, even though the system has 256 transmitters and receivers.

The next Resource specification is the speed of sound. The default is the average speed of sound in the human body of 1540 m/s, so this line is not really required unless a different speed of sound will be used, for example, if imaging in water. It is explicitly listed here to indicate that the speed of sound, if different from the default, should be defined at the top of the script, since it will be used to compute wavelengths as a unit of distance in the some of the structures below.

We will initially run our script in simulate mode, by adding the following line:

```
Resource.Parameters.simulateMode = 1;      % run script in simulate mode
```

This will allow us to check correct operation of our sequence without programming the hardware. In general, if a script runs correctly in simulate mode, it will run correctly with the hardware. Simulate mode ignores the presence of the hardware altogether, and uses software simulation to generate the RF data. In this case, VSX will not generate the extra object attributes required for programming the hardware, and sequence events will be executed only by the software sequencer.

To run in simulate mode, we need to define a Media model to generate ultrasound echoes. The Media model is simply a set of point targets that have a defined spatial location and reflectivity. The spatial location is defined in a three dimensional coordinate system tied to the transducer - in the case of the linear array L11-4v, the transducer elements are located along the x axis, centered around $x=0$, and the scan depth dimension is along the z axis. The y axis represents the elevational dimension, and is not used for a 2D linear scan format. For our example script, we will define a single media point target at a range of 100

wavelengths from the center of our transducer, with a reflectivity of 1.0. This requires only a single line of code, as follows.

```
% Specify media points  
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]
```

4.1 The Transducer (Trans) definition

The next structure to define in a setup script is the transducer that will be connected to the system. In this example, we will use the L11-4v linear array transducer, a probe used often for vascular imaging. If the transducer is one that is known by the software, which is the case with the L11-4v, its attributes can be defined using the *computeTrans.m* utility function, as shown below. [The *computeTrans.m* utility function is one of several utility functions that are provided to aid in setting various sequence object parameters. These utilities are written in the Matlab programming language, and can be inspected by the user if one needs a better understanding of their function.]

```
% Specify Trans structure array.  
Trans.name = 'L11-4v';  
Trans.frequency = 6.25; % not needed if using default center frequency  
Trans = computeTrans(Trans); % L11-4v transducer is 'known' transducer.
```

The *computeTrans.m* function generates all of the required attributes for the L11-4v transducer, including the position of the elements in the coordinate system for the probe. If one executes the above three lines at the Matlab command prompt, the full Trans structure will be created in the Matlab workspace, which can then be viewed with the Matlab workspace variable viewer. The attributes will be shown as below (additional attributes may also be present):

1x1 struct with 19 fields

Field ▲	Value	Min	Max
name	'L11-4v'		
units	'mm'		
lensCorrection	1.4785	1.4785	1.4785
frequency	6.2500	6.2500	6.2500
Bandwidth	[4.5000,10.5000]	4.5000	10.5000
type	0	0	0
id	109492	109492	109492
connType	1	1	1
numelements	128	128	128
elementWidth	0.2700	0.2700	0.2700
spacingMm	0.3000	0.3000	0.3000
ElementPos	128x4 double	-19.0500	19.0500
ElementSens	1x101 double	5.2718e-18	1.0000
impedance	53x2 complex double	3	1.7400e+01 - 1.2300e+02i
maxHighVoltage	50	50	50
spacing	1.2175	1.2175	1.2175
Connector	128x1 double	1	128
IR1wy	146x1 double	-0.0620	0.0471
IR2wy	146x1 double	-0.0369	0.0464

A change from earlier Verasonics software releases is that the default units of length are now output by `computeTrans` in mm rather than in wavelengths (with the one exception of `Trans.spacing`, which is still in wavelengths. This is indicated by the `Trans.units` attribute, which is set to 'mm'. To specify all length attributes in wavelengths, set `Trans.units = 'wavelengths'` before calling `computeTrans`. [Note: A custom probe `Trans` definition that was previously generated in wavelength units should have `Trans.units = 'wavelengths'` added to its set of attributes for proper functioning.]

One can specify a different center frequency from the default 6.25 MHz if desired, but this attribute should be set **before** calling `computeTrans`, since calculations of wavelengths will depend on this value. The Verasonics 3.x software and later allow setting Receive sample rates independently from the `Trans.frequency` specification, so there is little reason to change the default setting. The 'maxHighVoltage' attribute sets the maximum high voltage that the system will allow to be applied to the transducer (+/- maxHighVoltage). `Trans.maxHighVoltage` can be set by the user before or after calling `computeTrans`, since `computeTrans` will not overwrite this attribute.

With the transducer defined, we now need to define some storage space for the RF data that we want to acquire. This requires defining a Receive Buffer in the host computer (recall that the Receive Buffer is the only buffer that can receive RF data transferred from the Vantage local memory on the acquisition modules).

```
% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048; % allows for max depth of 256 wls
Resource.RcvBuffer(1).colsPerFrame = 128; % see text below
Resource.RcvBuffer(1).numFrames = 1; % minimum size is 1 frame.
```

The Receive Buffer will be created in the work space of Matlab, and will be accessible as a standard Matlab cell array (more on this later). The datatype assigned is 16 bit signed integers, which is currently the only RcvBuffer datatype supported. The dimensions of a RcvBuffer are (samples, channels, frames). The philosophy in defining a RcvBuffer is that a RcvBuffer frame can contain all the acquisition data needed for an entire frame, including all the multiple acquisitions needed to perform an image reconstruction. [The concept of an acquisition frame is arbitrary, and generally consists of all the acquisitions needed to form an image; however, the term can be applied to any set of acquisitions that one wishes to make prior to transferring the receive RF data to the host computer.]

The `rowsPerFrame` attribute is set according to the number of samples of RF data per channel that will be acquired, but typically is set to a larger size than required. Setting this parameter generally requires some calculation, based on the depth of the acquisition, the sample rate and the number of acquisitions per frame. If not set by the user, VSX will set the effective A/D sample rate to 4 times the transducer center frequency, as specified in `Trans.frequency`, which for 6.25MHz equates to 25MHz. (There are a finite number of A/D sample rates, determined by the 250MHz master clock of the system. If the sample rate is not specified, the software will find the closest frequency above the `Trans.frequency` value that yields a supported 4x sample rate.) At a sample rate of $4 \times \text{Trans.frequency}$, we have 4 samples per wavelength, so our setting of 2048 `rowsPerFrame` (samples) allows acquiring up to 512 wavelengths of data, for a maximum depth of 256 wavelengths (since the acquisition length is the round trip distance). The maximum number of samples for a single acquisition is limited only by the size of local memory on the Vantage acquisition modules, but for ultrasound imaging in the human body, most transducers can only acquire echo data out to about 512 wavelengths in depth (1024 wavelengths round trip), due to frequency dependent attenuation. [512 one way wavelengths at 6.25MHz and a speed of sound of 1540 m/s computes to 12.6 cm.] For this example, we will only acquire ultrasound data to a depth of 200 wavelengths, or 1600 samples round trip, which is well under our `rowsPerFrame` value of 2048 samples.

The `colsPerFrame` attribute is typically set equivalent to the number of receive channels available in the system. When data are transferred from the Vantage Unit's local memories to the host, the data from all receive channels are transferred together, with each channel transferred to its corresponding column of the Receive Buffer.

[Note: With a Vantage 64LE system, the `colsPerFrame` attribute is set to 128, even though only 64 channels of data can be transferred at a time. The 128 columns of the RcvBuffer correspond to the 128 connector I/O channels, so depending on the setting of the individual channel 2-1 multiplexers, the RF data will be transferred to one of two possible columns for each receive channel. The column not set to receive data will be cleared to all zero values.]

The `numFrames` attribute is used when multiple frames of RF data are to be acquired. The software can re-process the frames saved in the RcvBuffer and play back the processed frames in a cineloop. In this example, we are acquiring data from a single transmit, and one RcvBuffer frame is sufficient.

4.2 The Transmit Waveform (TW) definition

We can now define the attributes of the objects that will be used in our sequence of events. For our first desired action - “Transmit plane wave,” we need to define a transmit waveform, that will be used in a single transmit event.

```
% Specify Transmit waveform structure.  
TW(1).type = 'parametric';  
TW(1).Parameters = [6.25,0.67,2,1];    % A, B, C, D
```

While the Vantage system hardware can generate arbitrary transmit waveforms, for defining a simple transmit waveform we can use a ‘parametric’ definition, which involves specifying only four parameters, A, B, C and D. The A parameter is the frequency of the transmit pulse and sets the half cycle period of our waveform. The B parameter is used to set the amount of time that the transmit drivers are active in the half cycle period. We can set this value to between 0.1 and 1.0 to control the amount of power delivered by a transmitter. In this case, we use a value of 0.67 to generate a digital drive waveform that approximates a sine wave. The C parameter is the number of half cycles in the transmit waveform, which in this case is set to 2, providing a single cycle burst. Finally, the D parameter specifies the initial polarity of the first half cycle, with 1 indicating a positive polarity. By default, the drive waveform specified will have equalization pulses added to the start and end of the burst, so that the drive waveform to the transducer will look as in Fig. 4.1 below. The adding of the equalization pulses can be disabled, but this is not recommended as they insure that the overall waveform has no DC voltage component, even for odd numbers of half cycles. A DC component would create an initial offset in the receive signals that would decay slowly, adversely influencing the acquisition of signals near the transducer.

Note that if only a single transmit waveform has been specified, VSX will program the specified single waveform into all of the 128 transmitters. It would generally be inappropriate to have different A and C parameters over different transmitters, but one might want to modify the B parameters on individual transmit channels to achieve a form of transmit apodization. The modification of the B parameters for Transmit apodization can be performed automatically by VSX by setting the weighting values in the TX.Apod attribute of

a TX structure, to be discussed later.

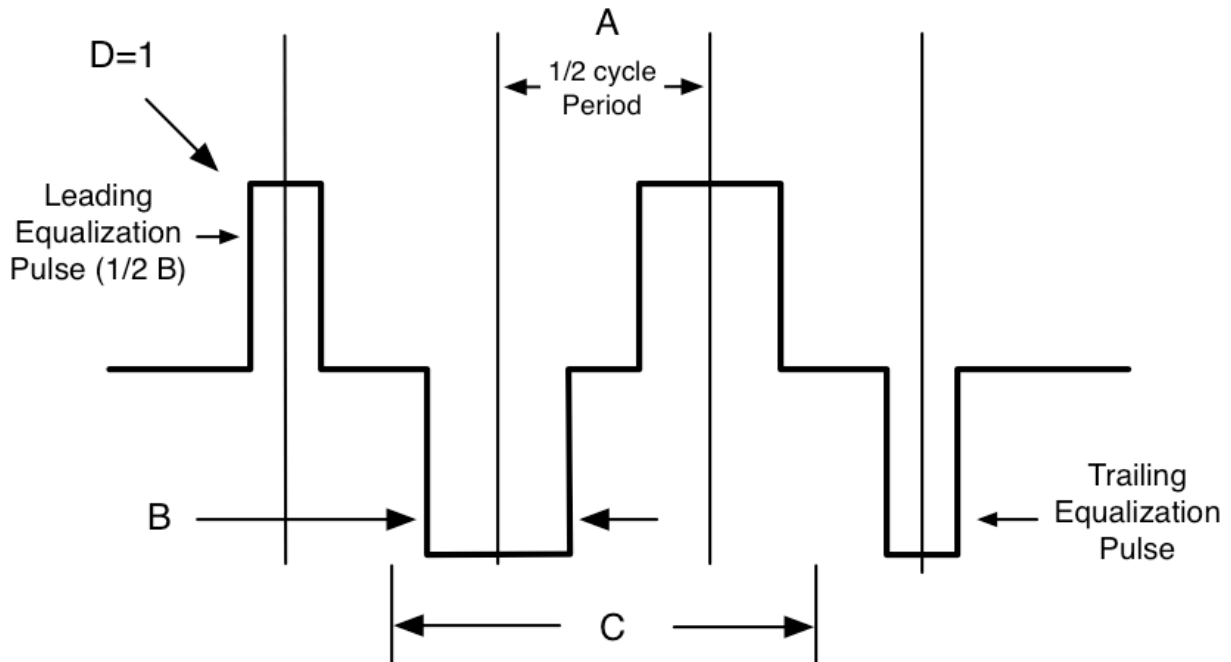


Fig. 4.1 Two half-cycle transmit pulse with equalization pulses added.

In the case where one really would like independent parametric waveforms on individual channels, the `TW.Parameters` array can be expanded to a 128 (or 256) row, two dimensional array, where each row specifies the waveform for the transmitter of the same number as the row index.

The Vantage system supports other methods for defining transmit waveforms, including the `TW.types` of 'envelope', 'pulseCode', and 'function'. These additional methods are described in the Sequence Programming Manual.

4.3 The TX object definition

Next we will define a transmit action using the TX structure. We need to define a TX structure for each unique transmit action in our sequence. Multiple transmits that have the same set of attributes can be specified with a single TX definition. The transmit waveform to use should be previously defined, and is referenced by the TX structure.

```
% Specify TX structure array.
TX(1).waveform = 1;           % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));
```

For our transmit action, we will use a plane wave excitation, where all of the elements transmit at the same time. This is indicated by setting `TX(1).focus = 0`, which the `computeTXDelays` utility function will detect as a special value, and set all of the 128 `TX.Delay` values to 0. The `computeTXDelays` utility function takes as input the `TX.focus` and `TX.Apod` values and computes transmit delay values for the active transmitting

elements of the array. The Trans structure must already exist in the Matlab workspace before calling *computeTXDelays*. The TX.Delay values are currently calculated in wavelengths of the transducer center frequency rather than time - one wavelength represents a time value equal to the period of Trans.frequency. For example, a TX.Delay(n) value of 10 at our example center frequency of 6.25 MHz is equivalent to 1.6 usecs.

The TX(1).Apod array is used to set which transmitters are active, and to specify a transmit apodization function, if desired. The length of TX.Apod is the smaller of the available aperture of the transducer or the number of transmitters (a transducer with a smaller number of elements than the number of transmitters requires a Trans.Connector specification, to identify which elements are connected to which connector I/O channels). A one for the value of TX(1).Apod(n) turns on the transmitter for element n and sets it to use the waveform defined in TW(TX(1).waveform). A value less than one for TX(1).Apod(n) will modify the B parameter of the transmit waveform for element n. The B parameter will be multiplied by TX(1).Apod(n) and rounded to the nearest value supported by the hardware (4 nsec resolution). This will implement a somewhat coarse form of transmit apodization, but one that can be effective in reducing transmit side lobes. In this example, we choose to have all of our transmitters operating at full power, so we set all the TX(1).Apod values to 1.

4.4 The TGC object definition

We now can address our next desired action - "Receive echo signals from plane wave transmit and store in local memory." For this action we need to program the analog signal amplifiers on each receive channel and specify how the signals are digitized, filtered and stored.

For programming the receiver amplifier gain, we need to define a TGC (Time Gain Control) waveform, that specifies the receiver gain as a function of time from the transmit burst. We will need the TGC waveform for our Receive specification, so it is a good idea to define it before it gets referenced. The definition is as follows:

```
% Specify TGC Waveform structure.  
TGC(1).CtrlPts = [500,590,650,710,770,830,890,950];  
TGC(1).rangeMax = 200;  
TGC(1).Waveform = computeTGCWaveform(TGC);
```

What is defined here is a set of eight control values that will be used to construct a TGC curve. The values range from 0 to 1023, where 0 is minimum gain and 1023 is maximum gain. The eight values represent the gain at increasing depth in our acquisition period, and are equally distributed over the range from 0 depth to TGC.rangeMax, whose value is in wavelengths. The actual TGC curve generated has sample points every 800 nsec, but the eight control points are sufficient to program the curve, as the slope changes fairly slowly. The various control values will be able to be changed during run time by means of a set of TGC sliders on the user interface window, so the values given here are used to set the starting position of the sliders. The utility function, 'computeTGCWaveform(TGC)', computes the actual TGC curve which will be sent to the hardware. The TGC curve is re-computed whenever a control slider changes, and the new curve is sent to the hardware.

4.5 The Receive object definition

Now we need to define a Receive operation that will specify other attributes of the input signal processing. The Receive structure specifies many parameters, including how long the receive period should run, which TGC waveform to use, how the received data are to be sampled and filtered, which receive channels are active, and where to store the RF data. A separate Receive structure is needed for every acquisition that is destined to be stored in a unique location in our Receive buffer. The structure is shown below.

```
% Specify Receive structure array -
Receive(1).Apod = ones(1, 128);
% For a Vantage 64 or 64LE, use these alternative statements for Receive.Apod
Receive(1).Apod = ones(1,64); % Vantage 64
Receive(1).Apod = [ones(1,64) zeros(1,64)]; % Vantage 64LE
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1; % Use the first TGC waveform defined above
Receive(1).mode = 0;
Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];
```

The first attribute, Receive(1).Apod is the apodization function for the transducer aperture. This array size is always the smaller of the available aperture of the transducer, or the number of connector I/O channels. For the L11-4v, the number of elements is 128, which matches the number of I/O channels available on the transducer connector. The Receive.Apod array values are used to set the scaling multiplier after the Input Filter (refer to Fig. 3.1) and the values can range from -4.0 to 4.0. In this case, we set all the apodization values to 1.0.

Note that for a Vantage 64 or 64LE system, only the first 64 channels are set to receive data with apodization values of 1.0. For the Vantage 64LE, the last 64 apodization values are set to zero, indicating that they are inactive. [Since only 64 channels are available for the 64LE, only 64 indices of the Receive.Apod array can be set to values other than zero, and generally these indices are contiguous. If more than 64 indices are set non-zero, an error will be generated when the script tries to run with the hardware. An error will also be generated if the 64 non-zero indices of the Receive.Apod array are set in a manner that is not supported by the 2-1 multiplexing. For example, indices 1 and 65 can not both have non-zero values.]

The next two attributes, startDepth and endDepth, set the start and end of A/D sampling, in units of wavelengths. Here we set the start of sampling at depth 0 and the end of sampling at 200 wavelengths (200 wavelengths at 6.25 MHz and sound speed of 1540m/s = 49.28 mm).

The index for the TGC curve to use for this receive period is set by the value of Receive(1).TGC. In this case the index is one, which references our previously defined

TGC waveform. One can defined multiple TGC waveforms if desired, and use a different waveform for different Receive structures.

The attribute `Receive(1).mode` determines the acquisition mode for the receive period. Here it is set to zero to indicate that new acquisition data replaces old data in the local memory for the receive channel. Other modes are 1 and 2, which are used to accumulate acquisition data in local memory (these modes are typically only used for very low signal-to-noise applications).

Next we define the desired storage location for our acquisition data in the Receive Buffer. This will be the destination for the acquisition data when it is transferred to the host computer. The temporary destination in the local memory of acquisition modules doesn't need to be specified, as it is automatically managed by the system software. Here we define `bufnum = 1`, the first (and only) `RcvBuffer`, `framenum = 1`, and `acqNum = 1`. The `acqNum` attribute is used to identify different acquisitions when there are multiple acquisitions that make up a frame. When multiple acquisitions are to be stored in a `RcvBuffer` frame, they are stored vertically down the columns of a `RcvBuffer` frame. Each column corresponds to a receive channel, so one channel's data will consist of a single column, with the multiple acquisitions stacked consecutively down the column.

The next attribute, `sampleMode = 'NS200BW'`, specifies that the RF signal will be Nyquist sampled at 200% bandwidth of the transducer center frequency. (Nyquist sampled means that the sample rate will be greater than two times the highest frequency in the signal.) Other `sampleMode` choices are `'BS100BW'`, `'BS67BW'` and `'BS50BW'`, which correspond to bandwidth sampling at signal bandwidths of 100%, 67% and 50%. The lower bandwidth choices require a corresponding bandpass filter setting for the `InputFilter`, to be described below. It is advisable to use the lowest `sampleMode` bandwidth setting appropriate for the signals to be received, along with the corresponding `InputFilter`, as this will lower DMA transfer bandwidth requirements and improve signal to noise. Since the bandwidth of the L11-4v transducer is around 100%, we will use the `'NS200BW'` `sampleMode` to capture the full spectrum of the transducer (the 100% bandwidth spectral points are typically at -6dB).

The `LowPassCoef` and `InputFilter` attributes can be used to program the Low Pass and Band Pass FIR digital filters that follow the A/D converter (refer to Fig. 3.2 above). These filters only apply when running with the hardware (not in simulate mode), but we will describe them here. The purpose of the Low Pass Filter is to eliminate frequencies higher than twice the transducer center frequency, so that the digital samples can be decimated (if necessary) to a 4 times center frequency, F_c , sample rate. (The A/D converter is typically programmed to sample at a rate sufficient to adequately sample frequencies up to the anti-aliasing filter cutoff (which can be set to different cutoffs). The digital Low Pass Filter can then eliminate high frequency noise that is above the bandwidth of the transducer and decimate to the $4 \times F_c$ sample rate, which is required if one wants to do image reconstruction processing.) The Band Pass Filter (`InputFilter`) is provided to shape the signal spectrum of the transducer - for example, one might want to narrow the spectrum for Doppler acquisitions, or filter out low frequency components for harmonic imaging.

The `LowPassCoef` attribute is an array of 12 coefficients that specify a 23 tap symmetric window function, with the 12th coefficient representing the center tap. The range for each coefficient is from -0.99997 to +1.0. If this attribute is missing or empty (as defined above),

a default set of coefficients will be programmed by VSX, taking into account the transducer frequency and setting the appropriate decimation rate.

The `InputFilter` attribute is an array of 21 coefficients (`c1 - c21`) that will be used by the Band Pass FIR filter following the A/D converter and Low Pass Filter. The coefficients define a symmetric window of a 41 tap FIR bandpass filter, with the 21st coefficient as the center tap coefficient. For a series of input samples `s0 – s41`, the output sample would be $(s0+s40)*c1 + (s1+s39)*c2 + (s2+s38)*c3 + (s3+s37)*c4 + \dots + (s19+s21)*c20 + s20*c21$. The coefficients can have a range of -0.99997 to +1.0. As with the `LowPassCoef`, if this attribute is missing or empty (as defined above), a default set of coefficients will be programmed by VSX, which provide a filter bandwidth consistent with the `sampleMode` attribute. It is desirable to have a band pass filter setting for most applications so that, at a minimum, we eliminate any DC component of our RF signals. In our example case with the L11-4v, the default Band Pass filter bandwidth with `sampleMode = 'NS200BW'` is considerably greater than the transducer bandwidth (around 100% for L11-4v), so there is no loss of signal.

4.6 The Event object definition

We are now ready to specify the sequence of events needed to acquire our data. Each event can specify multiple operations by referencing previously defined structures - a transmit operation (TX), usually paired with a receive operation (Receive), a reconstruct operation (Recon), a processing operation (Process) and a control operation (SeqControl). For our single acquisition, we need only a single Event specification. This Event will consist of a transmit and a receive operation, using our previously defined TX and Receive specifications, and a control operation to transfer the acquisition data from local memory on the Acquisition Modules to the RcvBuffer in the Matlab memory space of the host computer.

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;          % use 1st Rcv structure.
Event(1).recon = 0;        % no reconstruction.
Event(1).process = 0;      % no processing
Event(1).seqControl = 1;   % transfer data to host
    SeqControl(1).command = 'transferToHost';
```

The `Event(1).info` attribute is an optional string with whatever text the user wants to use to identify the event. This is helpful when looking at a list of events in the Matlab workspace viewer to help identify the operation of the event. The `Event(1).tx` and `Event(1).rcv` attributes specify the indices of our TX and Receive structures defined earlier. This Event has no reconstruction or processing operation, so the `Event(1).recon` and `Event(1).process` attributes are set to 0.

For simulation mode, the RF data is simulated directly to the RcvBuffer, but later we will want to acquire data with the hardware, which will be written to local memory on the VDAS modules. To implement our next to last sequence action - “Transfer individual channel data to host computer,” we need a SeqControl command called ‘transferToHost’. [The `Event(1).seqControl` attribute specifies the index of the SeqControl command, and we can

define the SeqControl structure immediately, which is done on the following line. For short sequences that don't contain loops, defining the SeqControl structures within the Event list is convenient; for longer sequences, the SeqControl structures can be pre-defined before the Event list is generated.] The 'transferToHost' command will move the **previous frame** of acquisition data acquired in the Event list from local memory to the RcvBuffer in host memory. For example, if the Event list specifies the acquisition of two frames of data before the 'transferToHost' command, only the second frame would be transferred. We therefore need a 'transferToHost' command for each frame of acquisition data acquired. It is ok to include the 'transferToHost' command in our simulation events, since this command is ignored by the software sequencer.

When all Events in a sequence have completed, our sequence will automatically stop and return control to Matlab. We therefore don't need to specify any additional actions. Finally, to save our simple acquisition sequence, we need to save all our defined structures to a .mat file that can be loaded by VSX. Typically, .mat files are saved in the MatFiles directory. This can be accomplished with the following line.

```
% Save all the structures to a .mat file.  
save('MatFiles/L11-4vAcquireRF');
```

Note that we have named our .mat file with a similar name to the setup file name. This will make it easier to keep the setup file associated with the .mat file.

5.0 Running the Program

For reference, the entire SetUpL11_4vAcquireRF.m setup file is listed below.

```
clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128; % no. of xmit chnls (V64LE,V128 or V256).
Resource.Parameters.numRcvChannels = 128; % change to 64 for Vantage 64 or 64LE
Resource.Parameters.connector = 1; % trans. connector to use (V256).
Resource.Parameters.speedOfSound = 1540; % speed of sound in m/sec
Resource.Parameters.simulateMode = 1; % runs script in simulate mode

% Specify media points
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L7-4 transducer is 'known' transducer.

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048; % this allows for 1/4 maximum range
Resource.RcvBuffer(1).colsPerFrame = 128; % change to 256 for V256 system
Resource.RcvBuffer(1).numFrames = 1; % minimum size is 1 frame.

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [6.25,0.67,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
TGC(1).CntrlPts = [500,590,650,710,770,830,890,950];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify Receive structure array -
Receive(1).Apod = ones(1,128); %V64, ones(1,64); V64LE [ones(1,64) zeros(1,64)];
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1; % Use the first TGC waveform defined above
Receive(1).mode = 0;
Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];

% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1; % use 1st TX structure.
Event(1).rcv = 1; % use 1st Rcv structure.
Event(1).recon = 0; % no reconstruction.
Event(1).process = 0; % no processing
Event(1).seqControl = 1; % transfer data to host
SeqControl(1).command = 'transferToHost';

% Save all the structures to a .mat file.
save('MatFiles/L11-4vAcquireRF');
```

When the setup file is executed, a file named 'L11-4vAcquireRF.mat' will be created in the Vantage software MatFiles directory or appropriate subdirectory (make sure your command line path is set to the Vantage software release directory. We can now run our script by typing

```
> VSX <cr>
```

Name of .mat file to process:

VSX asks for the name of the .mat file to process, so we add to the line as shown below.

```
Name of .mat file to process: L11-4vAcquireRF <cr>
```

If everything is working, we will see the GUI window in Fig. 4.1 below appear.

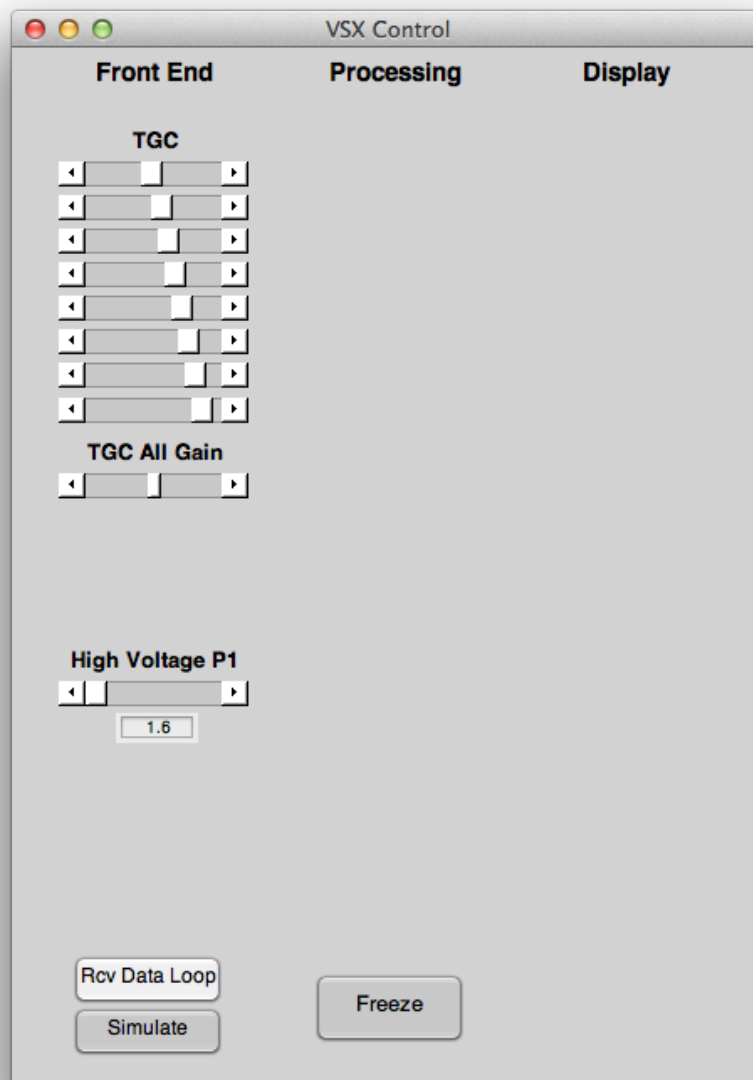


Fig. 4.1 Graphical User Interface (GUI) window for L11-4vAcquireRF.

What we see in the window are the default GUI controls that have been set by VSX and the `vsx_gui.m` function. The window is divided into vertical columns of controls, corresponding to 'Front End', 'Processing' and 'Display' functions. In the 'Front End' column, we see at the top controls for modifying the TGC curve, with the sliders set to the default positions specified in our setup file. Below the individual TGC sliders is a 'TGC All Gain' slider that moves the entire TGC curve up or down.

Below the TGC controls is the 'High Voltage' slider, which is used to change the high voltage output level of the transmitters. The P1 after the 'High Voltage' label indicates that we are using profile 1 of the Transmit Power Controller (TPC profiles will be covered later). The voltage listed here is the magnitude, and the actual transmit waveform varies between plus and minus this value. Note that the control setting is defaulted to a minimum setting, for safety reasons. Since our script is executing in simulate mode, the High Voltage control is also not active.

At the bottom of the left hand column are two toggle buttons that affect the operation of our script. The upper toggle button is labeled 'Rcv Data Loop' and its purpose is to switch a script from acquiring new data to reprocessing the data in the `RcvBuffer`. Its use will be covered at a later time. The lower button is the 'Simulate' button, which is indicated on. This button can activate or deactivate simulate mode while running with the hardware, but is set on by default in this case, since our script specified that it should be run in simulate mode.

The center column of the GUI window is normally reserved for processing controls. Our script basically doesn't define any processing, so the only control defined for this column is the 'Freeze' button, which is used to stop and restart a loaded script. When a script is loaded, the default state of the 'Freeze' button is off, so the script begins executing immediately after loading. In this case the 'Freeze' button is on, indicating that our script has completed execution. The system automatically enters a freeze state when it reaches the end of the list of Events, and there is no jump back to the beginning of the list.

























If our example script were to define 'Display' operations, there would be additional default controls presented in the right side display column for manipulation of various image parameters. We will see these controls appear in a later section when we add image reconstruction and display capability to our script.

So what actually happens when we execute VSX with our example script? VSX reads in the `.mat` file we specified, and examines the objects defined. VSX adds certain missing attributes to our structures, but since we specified to run in simulate mode, it didn't add parameters needed for programming the hardware. VSX then opened the GUI window by calling `vsx_gui.m`, and made the controls visible. Next, VSX entered a 'while loop' that checks for an exit condition, and if not set, calls the Matlab external function (Matlab external functions are referred to as mex functions) named 'runAcq'. The mex function 'runAcq' performs the work of our script, including loading and starting up the hardware sequencer, if running with the hardware, and executing software actions associated with Events. In this case, 'runAcq' simulated the RF data for our script from the specified Media, TW, TX, and Receive objects in our single Event, and wrote the simulated data to the `RcvBuffer`. When it reached the end of Events, runAcq set the 'Freeze' state and returned to Matlab.

For a script that reaches the end of the Event list and enters the 'Freeze' state, control returns to VSX, and VSX simply waits for the 'Freeze' toggle button to be pressed. When that occurs, VSX resets back to the starting event of the sequence, and calls runAcq again. So in this case, every time we unfreeze by pressing the 'Freeze' toggle button, our script runs once, and enters the freeze state again. To exit VSX and return control to the Matlab environment, we click on the close button of our GUI window. This will close the GUI window and set the exit condition for VSX, returning our command prompt in Matlab.

Let's now look at a few attributes that VSX added to our objects. After exiting our sequence, we can examine the attributes of objects in the Matlab Variable Editor, by double-clicking on the variable name in the Workspace window. For example, double-clicking on TW brings up the following information in the Variable Editor;

 1x1 **struct** with 24 fields

Field ▲	Value	Min	Max
 type	'parametric'		
 Parameters	[6.2500,0.6500,2,1]	0.6500	6.2500
 samplesPerWL	64	64	64
 equalize	1	1	1
 PulseCode	[2,-7,10,13,1;7,-13,10,7,1]	-13	13
 CumOnTime	0.1600	0.1600	0.1600
 Numpulses	4	4	4
 refchnum	1	1	1
 Bdur	0.2760	0.2760	0.2760
 refRelPulseWidth	0.5797	0.5797	0.5797
 estimatedAvgFreq	7.2464	7.2464	7.2464
 sysExtendBL	0	0	0
 Zload	25.9638 - 13.8507i	25.9638 - 13.8507i	25.9638 - 13.8507i
 Zsource	8.0000 + 63.7425i	8.0000 + 63.7425i	8.0000 + 63.7425i
 chlpk1V	0.0167	0.0167	0.0167
 integralPkUsec	[0.0240,-0.0280]	-0.0280	0.0240
 wvfmIntegral	0	0	0
 fluxHVlimit	892.8571	892.8571	892.8571
 TriLvIWvfm	215x1 double	-1	1
 Wvfm2Wy	215x1 double	-0.9284	0.8203
 Wvfm1Wy	215x1 double	-0.8974	0.9695
 simChNum	1	1	1
 numsamples	344	344	344
 peak	1.9844	1.9844	1.9844

Note that amongst other attributes, VSX has generated a TW.Wvfm2Wy attribute. This waveform is the round trip version of the transmit waveform (twice filtered by the transducer spectral response), sampled at 250MHz. It was simulated from our parametric waveform parameters, along with the transducer Trans parameters and is needed for simulating the RF signals returned from our Media points. We can plot this waveform by selecting the Wvfm2Wy attribute, and then clicking on the plot button in the Matlab header bar. The result is shown below:

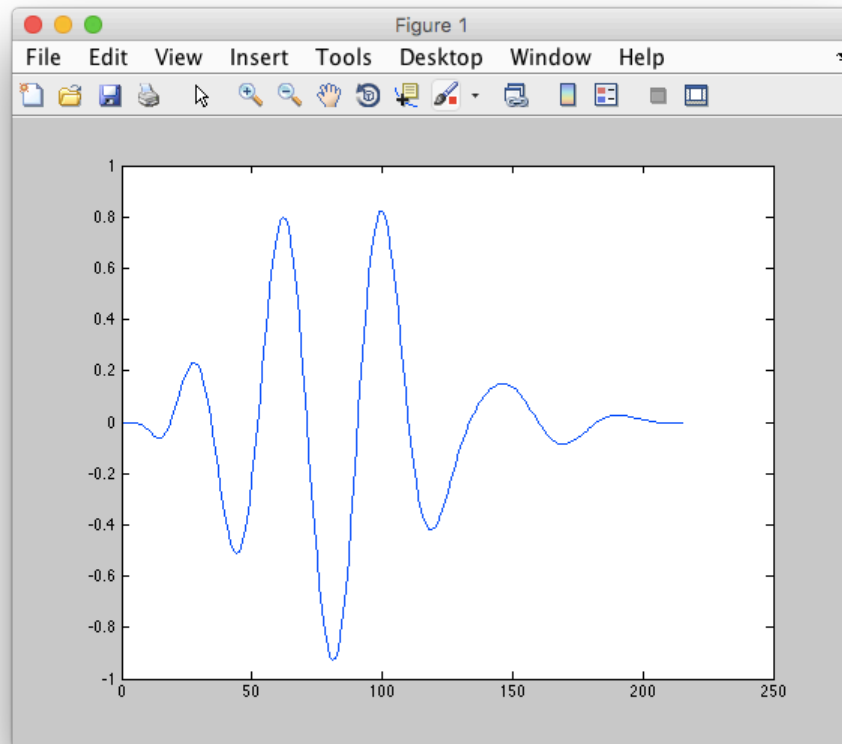


Fig. 4.2 Plot of simulated transmit waveform.

Next we examine the Receive structure in the Variable Editor.

1x1 struct with 18 fields			
Field ▲	Value	Min	Max
<input type="checkbox"/> Apod	1x128 double	1	1
<input type="checkbox"/> startDepth	0	0	0
<input type="checkbox"/> endDepth	208	208	208
<input type="checkbox"/> TGC	1	1	1
<input type="checkbox"/> mode	0	0	0
<input type="checkbox"/> bufnum	1	1	1
<input type="checkbox"/> framenum	1	1	1
<input type="checkbox"/> acqNum	1	1	1
<input checked="" type="checkbox"/> sampleMode	'NS200BW'		
<input type="checkbox"/> callMediaFunc	1	1	1
<input type="checkbox"/> decimSampleRate	25	25	25
<input type="checkbox"/> demodFrequency	6.2500	6.2500	6.2500
<input type="checkbox"/> samplesPerWave	4	4	4
<input type="checkbox"/> ADCRate	50	50	50
<input type="checkbox"/> decimFactor	2	2	2
<input type="checkbox"/> quadDecim	1	1	1
<input type="checkbox"/> startSample	1	1	1
<input type="checkbox"/> endSample	1664	1664	1664

Here we note that VSX has increased our `endDepth` value from 200 to 208. This is a result of the requirement that RF acquisitions be multiples of 128 samples. In this case, with 4 samples per wavelength, each wavelength of depth amounts to 8 samples (round trip) and consequently, our minimum depth increment for 128 samples is $128/8 = 16$ wavelengths. Since our specified depth of 200 samples amounts to 12.5 increments, VSX rounded up to 13 increments, giving 208 samples.

We also find two new attributes that were added by VSX - `startSample` and `endSample`. The `startSample` and `endSample` attributes identify the starting row and ending row indices in a channel column of the `RcvBuffer` data. Since a column may contain multiple acquisitions with different `acqNum`s, these indices help us to find the data that belongs to a specific `acqNum`. As we will see when we run our script with the hardware, there will be additional hardware specific attributes added by VSX.

We can now take a look at the actual RF data generated by our script. The Matlab array that is created for the `RcvBuffer(s)` is named `RcvData`, and is a cell array that has a cell for each `RcvBuffer` defined, with dimensions of (samples, channels, frames). In this case, there is only one `RcvBuffer`, corresponding to the `RcvData{1}` cell. To plot the RF samples for a specific channel, say channel 64, we can use the Matlab statement:

```
> plot(RcvData{1}(Receive(1).startSample:Receive(1).endSample,64,1)) <cr>
```

The above statement uses the `startSample` and `endSample` from our `Receive` structure to plot just the acquisition data for that `Receive`. (In this case, there is only one `Receive` for the column.) The plot is shown in Fig. 4.3 below.

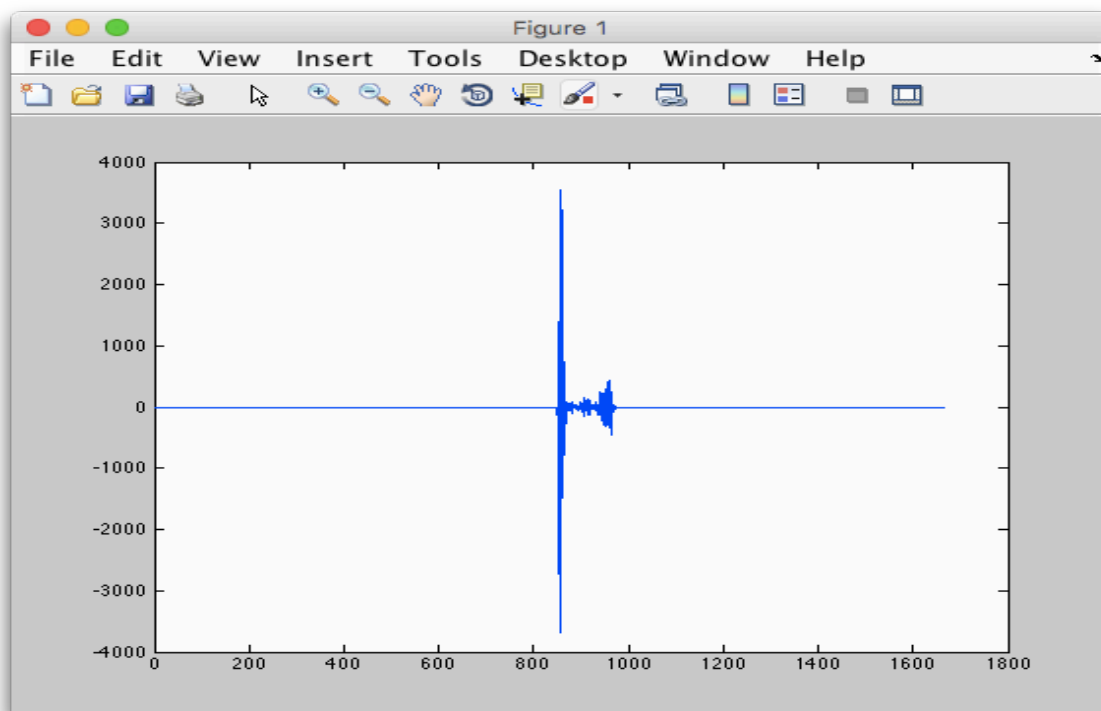


Fig. 4.3 Plot of RF data for channel 64.

In the plot we see a single large echo return, which is the echo from our single Media point, that we placed at 100 wavelengths beneath the center of our transducer array. We can zoom in on this signal, to get a better look.

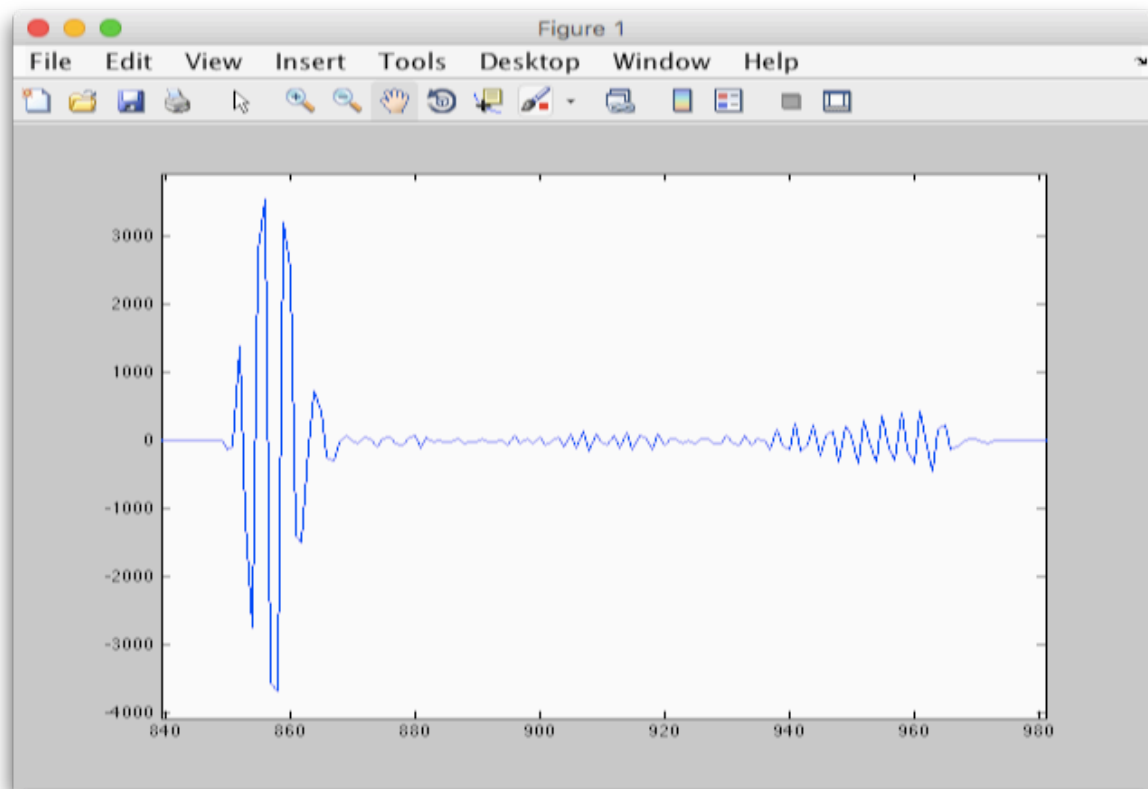


Fig. 4.4 Zoomed in plot of echo return for channel 64.

We see that the main echo return is very similar to the transmit waveform, but sampled at four times the center frequency rather than the 250MHz of the TW.Wvfm2Wy. This would be the expected return from a point target with our plane wave transmit. The small signals following the main echo are due to the fact that our plane wave transmit aperture is not infinite in extent, and the edges of the aperture contribute components that are not fully cancelled. An interesting experiment that can be performed entirely in simulate mode would be to investigate various transmit apodization functions that can reduce these edge effects for plane wave transmits.

Another way to look at our RF data set is to show the samples for all channels as a grayscale image. For this plot we can use the Matlab function, `imagesc`, which will scale the amplitude data to the range of grayscale intensities available. The Matlab commands are as follows:

```
> imagesc(RcvData{1})(Receive(1).startSample:Receive(1).endSample,:,1));
> colormap('gray')
```

The resulting image is shown in figure 4.5 below. (A Vantage 64 system will only show data for channels 1-64.)

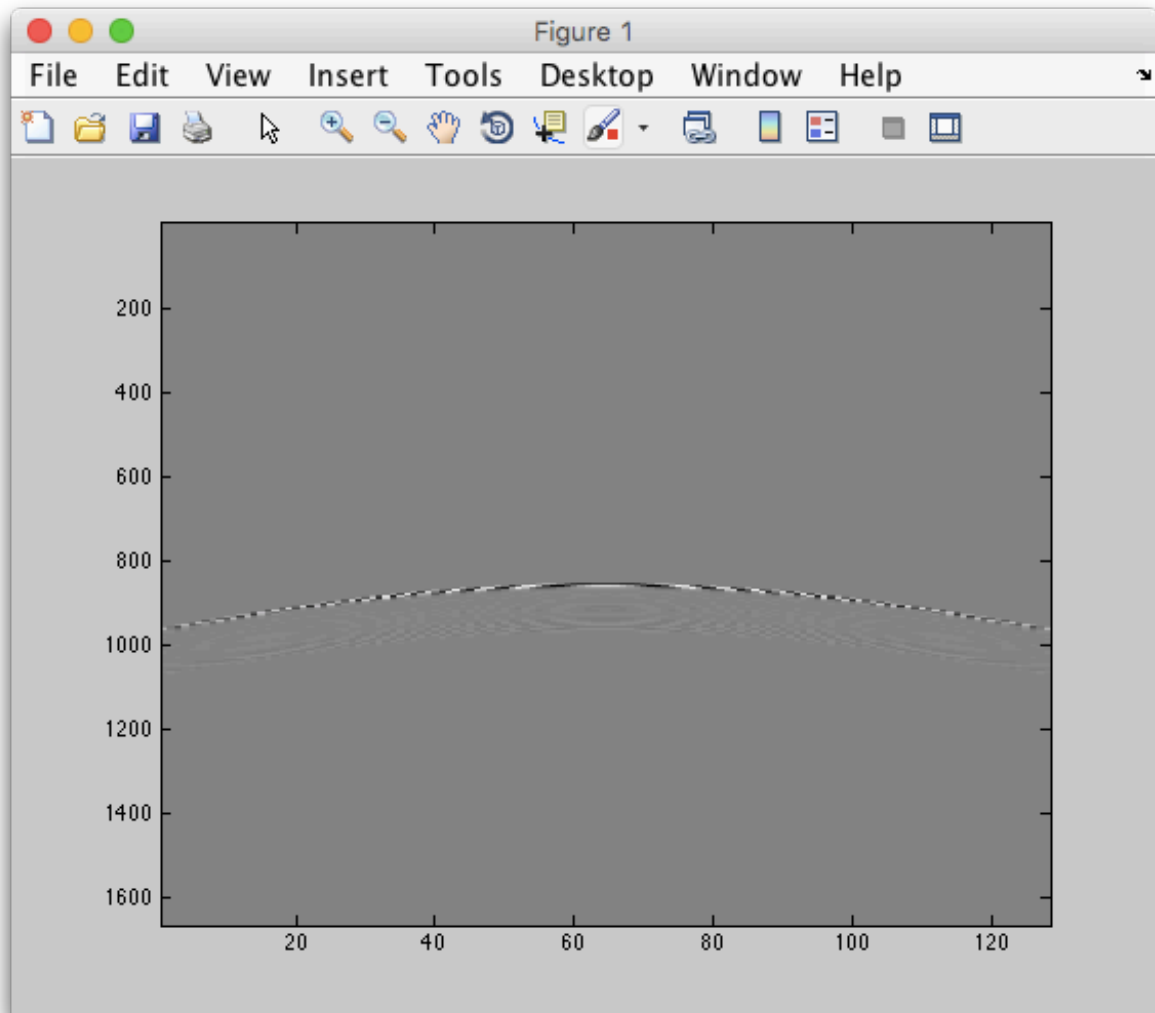


Fig. 4.5 Image of RF data - channels 1 - 128

The curvature of the main echo return in Fig. 4.5 is due to the increasing distance of the point target from the transducer elements as we move from the center of the aperture to the edge. Zooming in on the central portion of Fig. 4.5, we can see the peaks and troughs of the transmit waveform, as well as the clutter from the edge effects of the transmit aperture.

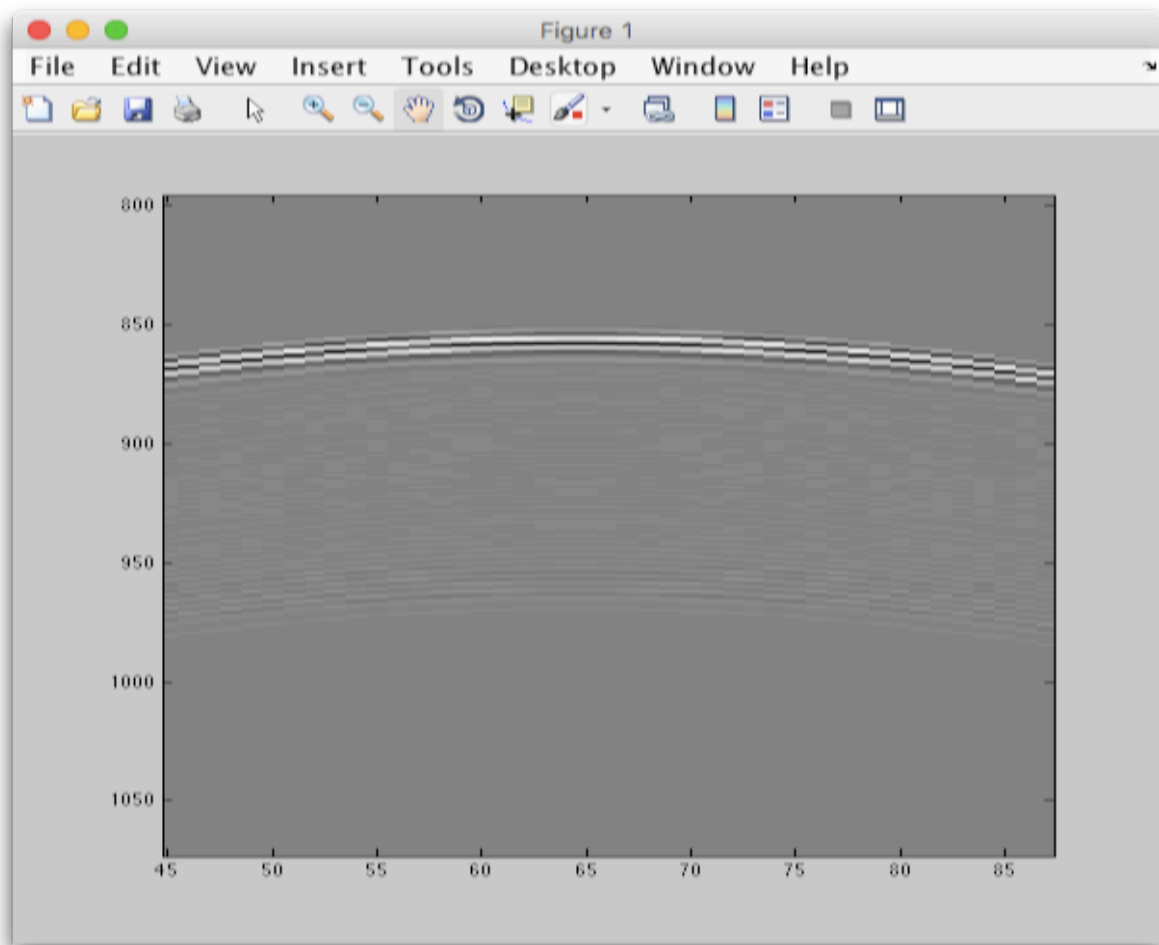


Fig. 4.6 Zoomed in central portion of Fig. 4.5

6.0 Acquiring multiple acquisition data with the Vantage system

In this section we will discuss how to acquire multiple acquisitions that make up a frame of receive data. Multiple acquisitions are needed for synthetic aperture scanning, Doppler sensing, and ray line imaging with focused transmits. The concept of a frame of data is somewhat arbitrary, but typically means all the data required to reconstruct an image of some sort for display. Ultrasound, being a real-time modality, generally aims to generate a number of image frames per second.

In the Vantage system, we would typically like to acquire complete frames of RF data into local memory on the Acquisition Modules before initiating a transfer to the host computer. A practical reason for this grouping of acquisitions is to improve the efficiency of the DMA transfers of acquisition data stored in local memory of the Acquisition Modules to the host computer RcvBuffer memory. There is an overhead associated with setting up and launching a DMA, and if we transferred each individual acquisition's data as they are acquired, the overall DMA transfer rate would be quite low. So to maximize DMA bandwidth, we need to transfer large amounts of data with a relatively low number of transfers. If we only transfer acquisition data at our intended frame rate, which in most cases is in the range of 10 to 60 frames per second, we can keep the DMA overhead down to a few percent of the available transfer time.

Another reason for transferring complete frames of acquisition data is for processing efficiency. To maximize efficiency, sequences can be arranged to acquire data for the next frame while the previous or a prior frame is being transferred to the host. Also, the processing of a frame of data can be arranged so that processing of a previously transferred frame can be initiated while a new frame is being transferred. This pipelining of acquisition, DMA transfers, and processing allows each of these actions to utilize the entire period between frames to accomplish their tasks, as shown in Fig. 6.1 below. Pipelining requires multi-frame storage capability in both local memory and in the RcvBuffer. The Vantage system provides large amounts of local memory (typically 48 MBytes or more for each channel), and the RcvBuffer memory is limited only by available memory in the host computer.

A third reason for frame based acquisition, transfer and processing is to simplify system operations, allowing easier automation of processes. For example, the 'transferToHost' SeqControl command automatically programs a DMA transfer action, collecting all the acquisitions into local memory for the previous frame of Receives, sending the data to the proper locations in the host RcvBuffer. Another example is the pixel-based reconstruction processing of RcvBuffer data. With all of the acquisitions collected for the frame, the reconstruction processing can be automatically divided up between multiple processing cores, which provides for significantly faster reconstruction processing.

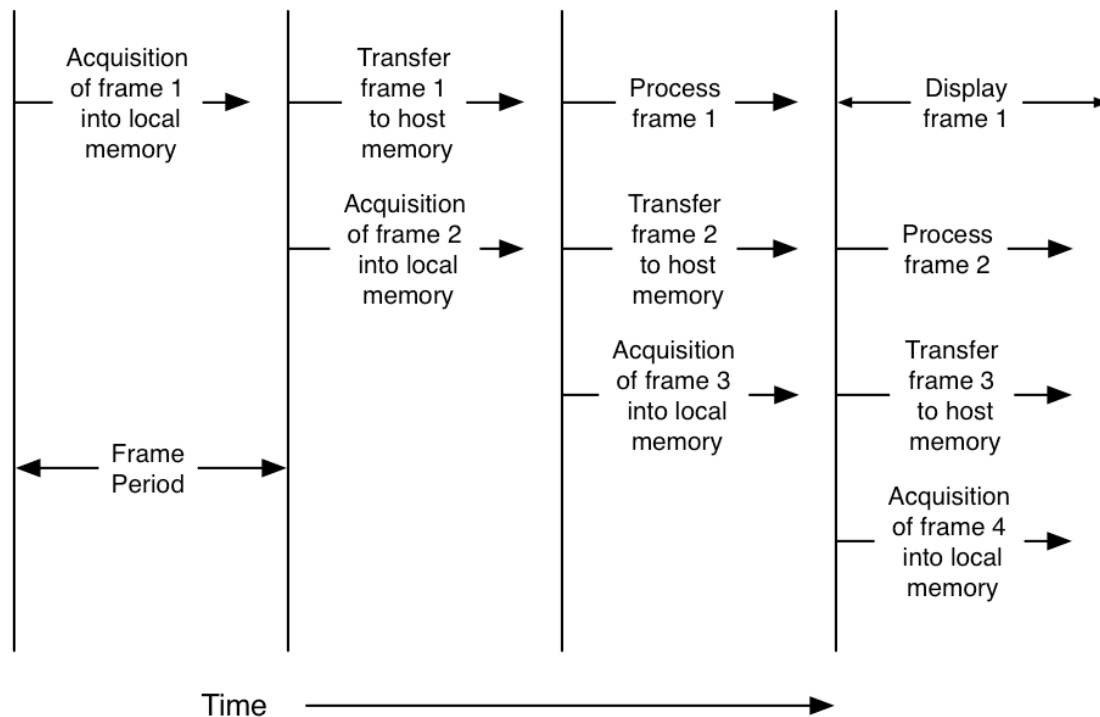


Fig. 6.1 Pipelining of acquisition, data transfer, and processing.

Let's now proceed to add another acquisition to our RcvBuffer frame. For this example, we will acquire the received signals over the full aperture of the transducer in two acquisitions. This approach would be required with a Vantage 64 system where only 64 receive channels are available and we want to 'synthesize' the full aperture on receive. (The extra acquisition is obviously not needed for a Vantage 128 or 256 channel system, but the example serves to illustrate how multiple acquisitions are stored and processed.) To accomplish this, we define a Receive(1).Apod array that specifies receiving on the first 64 elements of transducer array and then add another event that receives on the second half of the aperture. Our Receive structures are given below.

```
% Specify 1st Receive structure array -
Receive(1).Apod = zeros(1, 128);
Receive(1).Apod(1:64) = 1;
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1;
Receive(1).mode = 0;
Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];

% Specify 2nd Receive structure array -
Receive(2).Apod = zeros(1, 128);
Receive(2).Apod(65:128) = 1;
Receive(2).startDepth = 0;
Receive(2).endDepth = 200;
Receive(2).TGC = 1;
Receive(2).mode = 0;
Receive(2).bufnum = 1;
Receive(2).framenum = 1;
```



```

Receive(2).acqNum = 2;
Receive(2).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(2).InputFilter = [];

```

The second Receive structure is only different from the first in two attributes -

Receive(2).Apod and Receive(2).acqNum. The Receive(2).Apod array is set to have ones in indices 65 to 128, corresponding to the second half of the transducer aperture. The Receive(2).acqNum is set to 2, indicating that this is the second acquisition in the frame.

Here we emphasize an important rule. When defining Receive operations, define all the acquisitions for a frame with consecutively increasing structure indices and acqNums. This order of definition is necessary for the automatic programming of the DMA transfers. When VSX parses the Receive structures for operation with the hardware, it assigns storage locations in local memory in ascending order, so that all of the acquisitions for a frame are contiguous. This allows a single DMA transfer to move all the acquisitions for a frame to host memory.

With this added Receive definition, our sequence appears as follows:

```

clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128;           % no. of transmit channels.
Resource.Parameters.numRcvChannels = 128;        % change to 64 for Vantage 64 system
Resource.Parameters.speedOfSound = 1540;         % speed of sound in m/sec
Resource.Parameters.simulateMode = 1;            % runs script in simulate mode

% Specify media points
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans = computeTrans(Trans); % L7-4 transducer is 'known' transducer.

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 4096; % this allows for 2 acquisitions
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 1; % minimum size is 1 frame.

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [6.25,0.67,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numElements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
TGC(1).CntrlPts = [500,590,650,710,770,830,890,950];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify 1st Receive structure array -
Receive(1).Apod = zeros(1, 128);
Receive(1).Apod(1:64) = 1; % Receive on 1st half of transducer
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1;
Receive(1).mode = 0;

```

```

Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];

% Specify 2nd Receive structure array -
Receive(2).Apod = zeros(1, 128);
Receive(2).Apod(65:128) = 1; % Receive on 2nd half of transducer
Receive(2).startDepth = 0;
Receive(2).endDepth = 200;
Receive(2).TGC = 1;
Receive(2).mode = 0;
Receive(2).bufnum = 1;
Receive(2).framenum = 1;
Receive(2).acqNum = 2;
Receive(2).sampleMode = 'NS200BW';
Receive(2).LowPassCoef = [];
Receive(2).InputFilter = [];

% Specify sequence events.
Event(1).info = 'Acquire RF Data for 1st half of aperture.';
Event(1).tx = 1; % use 1st TX structure.
Event(1).rcv = 1; % use 1st Rcv structure.
Event(1).recon = 0; % no reconstruction.
Event(1).process = 0; % no processing
Event(1).seqControl = 0; % no SeqControl

Event(2).info = 'Acquire RF Data for 2nd half of aperture.';
Event(2).tx = 1; % use 1st TX structure.
Event(2).rcv = 2; % use 1st Rcv structure.
Event(2).recon = 0; % no reconstruction.
Event(2).process = 0; % no processing
Event(2).seqControl = 1; % transfer data to host
SeqControl(1).command = 'transferToHost';

% Save all the structures to a .mat file.
save('L11-4vAcquireRF');

```

Note that we have also added another Event to our sequence. The first Event transmits on the full aperture and acquires receive data on the first half of the aperture, while the second Event transmits on the full aperture again, acquiring receive data on the second half of the aperture. The 'transferToHost' SeqControl command has been moved to the second Event, and will transfer data from both acquisitions to host memory. More specifically, the 'transferToHost' SeqControl command creates a DMA transfer action after the Event that contains the command that transfers *all* the previous Receive data having the same RcvBuffer number and frame number. The acqNums in the frame will be transferred in order, and the acquisition data for each receive channel will be written sequentially down the RcvBuffer columns.

A change in our setup file which may have escaped notice is that the rowsPerFrame attribute of our RcvBuffer has been doubled to 4096. The number of rows in the RcvBuffer must be greater than the sum of all the samples in all the acquisitions for the frame. With an endDepth in both acquisitions of 200 wavelengths, and 4 samplesPerWave, we need at least 3200 rows in our RcvBuffer (200 wavelengths * 2 for round trip * 4 samplesPerWave * 2 acquisitions). [Note: When converting wavelengths of depth to samples, the result is always rounded up to the next 128 sample boundary, since the hardware acquires

samples in 128 sample blocks. This action changes our formula as follows: $128 * \text{ceil}(200 * 2 * 4 / 128) * 2 = 3328$ samples] Defining the `RcvBuffer.rowsPerFrame` larger than actually needed causes no problem, except for using some amount of additional memory. In sequences that change range, the `rowsPerFrame` attribute is usually set to accommodate the maximum depth of acquisition.

We can now re-run the setup file to re-generate the .mat file. Then we can again execute our script with VSX.

```
> VSX <cr>
Cleaning up memory allocated.
Name of .mat file to process: L11-4vAcquireRF <cr>
```

Again we see the GUI window appear and the system enter the freeze state. Closing the GUI window, we can again examine the RF data for all channels using `imagesc`.

```
> imagesc(RcvData{1}(:, :, 1));
```

The resulting image is shown below in Fig. 6.2.

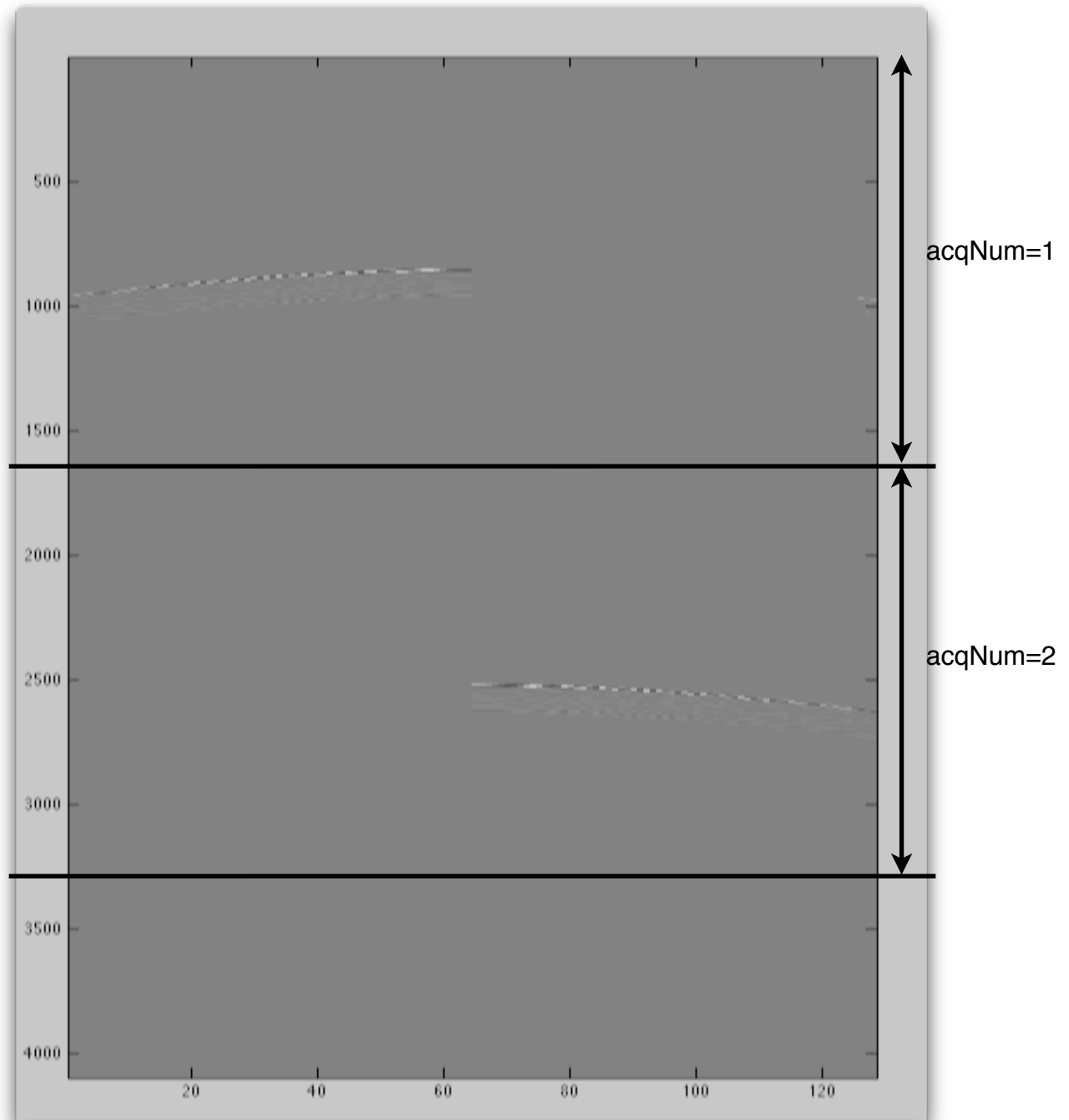


Fig. 6.2 RcvData for 128 channels with two acquisitions.

Here we can see the data for both of our acquisitions, which are stacked sequentially down the RcvBuffer column. The first acquisition runs from sample one to sample 1664 and the second acquisition runs from sample 1665 to 3328. The first acquisition (acqNum=1) shows receive data in only the first 64 channels, since we disabled the 2nd half of the aperture with Receive(1).Apod only having ones for elements 1-64. The second acquisition

shows receive data in only channels 65-128, since these again are the active channels specified in `Receive(2).Apod`.

With more complex scanning sequences, such as those using multiple focused ray lines with multiple focal zones, there may be hundreds of acquisitions in a frame, requiring hundreds of `Receive.acqNum` values. In these cases, the DMA transfers can take some number of milliseconds (the maximum DMA transfer rate of the Vantage system is over 6 GBytes/sec), and for maximum frame rate, it is necessary to “pipeline” the acquisition and DMA transfers. This is actually easier to achieve than it sounds, by using a multi-frame `RcvBuffer` and asynchronous processing, to be covered in a later section.

7.0 Using an External Function

In the previous sections, we have shown how to look at acquisition data in the Matlab workspace after exiting our sequence program. But this requires exiting our program and typing commands on the Matlab command line. If we know what data we want to process and/or display, it would be convenient to incorporate the Matlab commands into our sequence program. There is a way to do this, and it happens to be one of the most powerful features of the Vantage Research System. The approach is to define an external function, which executes in the Matlab workspace, that gets called automatically at a specified event in the Event list.

The external function call is implemented by defining an 'External' process class, using a Process object. The general form of a Process object is as follows:

```
Process =  
    classname    [string]    Class of processing object.  
    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.
```

The 'External' Process class is defined as:

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

In this class, the 'processFunctionName' provided for the method attribute is the name of a Matlab function that can be found on the Matlab directory path. The Parameters are a cell array of attribute/value pairs that specify various inputs and outputs for our function.

We will now create an external processing function that plots the RF data for a specific acquisition event and a specific channel, similar to the plot in Fig. 4.3 above. We will name our function 'myProcFunction', and specify as input the RcvBuffer defined in our script. There is no need to put data back into the system, so we don't need an output destination. Our Process structure is then defined as follows:

```
Process(1).classname = 'External';  
Process(1).method = 'myProcFunction';  
Process(1).Parameters = {'srcbuffer', 'receive', ... % name of buffer to process.  
                        'srcbufnum', 1, ...  
                        'srcframenum', 1, ...  
                        'dstbuffer', 'none'};
```

For the `srcbuffer`, we specify `receive`, and follow with the buffer number and frame number. The destination buffer, `dstbuffer`, is none. Our Process definition can be put just about anywhere in our script, but it is convenient to place it just after the Receive definitions and before the Event list.

We can now proceed to write our external function as a standard Matlab function in the Matlab environment. In a new editor window, we type the following commands.

```
function myProcFunction(RData)
persistent myHandle
% If 'myPlotChnl' exists, read it for the channel to plot.
if evalin('base','exist('myPlotChnl','var')')
    channel = evalin('base','myPlotChnl');
else
    channel = 32; % Channel no. to plot
end
% Create the figure if it doesn't exist.
if isempty(myHandle) || ~ishandle(myHandle)
    figure;
    myHandle = axes('XLim',[0,1500],'YLim',[-16384 16384], ...
                    'NextPlot','replacechildren');
end
% Plot the RF data.
plot(myHandle,RData(:,channel));
drawnow
return
```

The first line of our function provides the name and input variable for our function, `RData`. Next we define a persistent variable, `'myHandle'`, which will keep the handle to our plot window available over multiple calls to our function. The next if-else construct looks for a channel number defined in the main workplace, which will be used to define the channel to plot. If the variable is not defined, a default value of 32 is assigned. We will see how this can be used later. The next if statement checks to see if our plot window has been created, and if not, calls the Matlab `'figure'` statement to create it, and sizes the axes appropriately. To clear and update the plot portion of the figure, we use the `'NextPlot'`, `'replacechildren'` attribute pair. Finally, we can plot our new acquisition data with the `'plot'` statement, which we follow with a `'drawnow'` statement to make the plot appear. The range of the Y axis has been set to the full acquisition range of the digital data, which for Vantage is 15 bits (14 bits from the A/D converter and one bit of word growth from the digital filters). The `myProcFunction` function should be saved to the Matlab Simulator main directory or a directory available in the current path.

We return to our first example script, which uses a single transmit/receive acquisition and add the Event structure below to specify when to call the external function in the list of Events. For processing our acquired data, we place the call after the `'transferToHost'` command in its own Event.

```
Event(2).info = 'Call external Processing function.';
Event(2).tx = 0; % no TX structure.
Event(2).rcv = 0; % no Rcv structure.
Event(2).recon = 0; % no reconstruction.
```

```
Event(2).process = 1;      % call processing function
Event(2).seqControl = 0;
```

We list the revised sequence below for reference.

```
clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128;      % no. of transmit channels
Resource.Parameters.numRcvChannels = 128;   % change to 64 for Vantage 64 system
Resource.Parameters.connector = 1;          % trans. connector to use (V 256).
Resource.Parameters.speedOfSound = 1540;    % speed of sound in m/sec
Resource.Parameters.simulateMode = 1;        % runs script in simulate mode

% Specify media points
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L11-4v transducer is 'known' transducer.

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048; % this allows for 1/4 maximum range
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 1; % minimum size is 1 frame.

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [6.25,0.67,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
TGC(1).CntrlPts = [500,590,650,710,770,830,890,950];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify Receive structure array -
Receive(1).Apod = ones(1,128); % if 64ch Vantage, = [ones(1,64) zeros(1,64)];
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1; % Use the first TGC waveform defined above
Receive(1).mode = 0;
Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];

% Specify an external processing event.
Process(1).classname = 'External';
Process(1).method = 'myProcFunction';
Process(1).Parameters = {'srcbuffer','receive',... % name of buffer to process.
                        'srcbufnum',1,...
                        'srcframenum',1,...
                        'dstbuffer','none'};

% Specify sequence events.
```



```

Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;          % use 1st Rcv structure.
Event(1).recon = 0;        % no reconstruction.
Event(1).process = 0;      % no processing
Event(1).seqControl = 1;   % transfer data to host
    SeqControl(1).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;           % no TX structure.
Event(2).rcv = 0;          % no Rcv structure.
Event(2).recon = 0;        % no reconstruction.
Event(2).process = 1;      % call processing function
Event(2).seqControl = 0;

% Save all the structures to a .mat file.
save('L11-4vAcquireRF');

```

After executing the SetUp file to save the structures, we can run our sequence again with VSX. This time, we will see a new window open at the top of our screen that looks similar to Fig. 4.3. We see not quite all of the data from the acquisition in our script, since we specified that the plot only run to 1500 samples in myProcFunction.

8.0 Running with Acquisition Hardware

With our sequence executing properly in simulate mode, we can now switch over to acquiring actual RF data from a transducer. If you have access to an L11-4v linear array transducer, you can plug it into the single connector on a 128 channel Vantage system or the left hand connector (connector 1) on a 256 channel system. (If you don't have an L11-4v, the script can be run in fake scanhead mode with the connector open - you will have to agree to this when you run the script with VSX, which will detect that no transducer is connected. You will be able to see the ring down from the transmit and the noise floor with sufficient TGC receiver gain.) To switch over to acquiring with the hardware, we need to turn off simulate mode. Set

```
Resource.Parameters.simulateMode = 0;    % no simulate mode
```

or eliminate the line altogether.

Typically, the only difference in a script that runs in simulate mode and then is modified to run with the hardware is the `Resource.Parameters.simulateMode` attribute. However, there are some differences in behavior that need to be understood. We will digress for a moment to explain more detail on the two sequencers in the system.

8.1 Operation of the Hardware and Software Sequencers

As mentioned earlier, there are separate sequencers for the hardware and software that can run independently, and that operate differently on certain sequence events. The exact same series of Events is sent to each sequencer by VSX, but each sequencer only responds to event actions that it knows or cares about. For example, a transmit and receive action in an Event is only acted upon by the hardware sequencer (except in simulate mode, where the software simulates a transmit and receive action). When running with the hardware, the software sequencer simply ignores transmit and receive actions. Similarly, the hardware sequencer knows nothing about reconstruction and processing actions, and simply ignores those parts of an Event. Table 8.1 below shows the actions that each sequencer responds to. A check mark means that the sequencer processes the action, and a dash means the action requested is ignored.

Both the hardware and software sequencers execute the actions requested in the Event list at the rate at which the actions can be processed. This means that the sequencers rarely are executing the same Event, as each sequencer is processing actions at its own pace. This asynchronous operation of the sequencers is actually the desired mode of operation for most user scripts, but there are occasions when some synchronizing of the two sequencers is needed. For example, when acquisition is triggered by an external trigger source, the hardware sequencer stops and waits for the trigger at a specified Event in the sequence. In this situation, we would like the software sequencer to not process any acquisition data until the data are actually acquired. This requires halting the software sequencer until the acquisition data become available.

Let's take a brief look at some of the mechanisms for synchronizing the hardware and software sequencers.

Table 8.1 Hardware and Software Sequencer Actions.

Action	Hardware Sequencer	Software Sequencer
Event -> tx/rcv	✓	Only in simulate mode 1
Event -> recon	-	✓
Event -> process	-	✓
Event->seqControl.command		
‘call’	✓	✓
‘cBranch’	✓	✓
‘DMA’	✓	-
‘jump’	✓	✓
‘loopCnt’	✓	✓
‘loopTst’	✓	✓
‘markTransferProcessed’	-	✓
‘noop’	✓	Only in simulate mode 1 or 2, unless specified in condition.
‘pause’	✓	-
‘returnToMatlab’	-	✓
‘rtn’	✓	✓
‘setTPCProfile’	✓	-
‘stop’	✓	✓
‘sync’	✓	✓
‘timeToNextAcq’	✓	Only in simulate mode 1 or 2
‘transferToHost’	✓	-
‘triggerIn’	✓	-
triggerOut’	✓	-
‘waitForTransferComplete’	-	✓

Synchronizing using the 'sync' command - A simple mechanism for synchronizing the software and hardware sequencers is available that can be used at any event in a sequence. A SeqControl command call 'sync' can be placed in an event at the point in a sequence where the hardware and software sequencers are to be synchronized. When the hardware sequencer reaches an event with a 'sync' command, it will perform any acquisition actions first, and check for whether the software sequencer has already arrived at the same event. If the software sequencer has already arrived and is waiting for the hardware sequencer, the hardware sequencer will release the software sequencer from its paused condition and continue to execute following events. If the software sequencer has not yet arrived at the event with the 'sync' command, the hardware sequencer will pause, waiting for the software sequencer to arrive at the event. When the software sequencer arrives, both sequencers will then continue on to process subsequent events. From the software sequencers perspective, when it arrives at an event with a 'sync' command, it first performs the event's reconstruction and processing actions, then checks to see if the hardware sequencer has already arrived at the event and is waiting on the software sequencer. If so, the software sequencer releases the hardware sequencer from its paused condition and both sequencers continue on to process subsequent events. If the hardware sequencer has not yet arrived at the event, the software sequencer first cancels the pause at the event for the hardware sequencer, and then starts a polling loop to determine when the hardware sequencer arrives. When the arrival is detected, the software sequencer continues on to process subsequent events. When the hardware sequencer arrives at a sync point after the software sequencer has released the pause, the hardware sequencer continues on to the next event as if the sync point wasn't there, thus providing deterministic timing.

Synchronizing around DMA transfers - Since the main interaction of the hardware with the software is in the DMA transfer of acquisition data, some synchronizing mechanisms are built around these transfers. For each unique 'transferToHost' action, there is an associated internal flag, which we will call the 'Transferred, Not Processed' (TNP) flag. The flag is set by the hardware, and reset by the software. When the hardware performs a DMA transfer of a frame of data, as specified by a 'transferToHost' SeqControl command, it sets the TNP flag for that transfer at the end of the DMA transfer. (The flag may be already set, in which case it is left in the set condition.) If we want to pause the software sequencer to wait for a specific DMA transfer to occur, we can use the 'waitForTransferComplete' SeqControl command to program a wait for the specified transfer's TNP flag to be set. In an event that has the 'waitForTransferComplete' SeqControl command, any reconstruction or processing action will be delayed until after the TNP flag becomes set.

Note: The 'waitForTransferComplete' command is not needed when a reconstruction is specified in an event (Event.recon = xx) that takes place after a prior 'transferToHost', ***provided*** the reconstruction references the RcvBuffer frame that the transfer went into. This is because reconstruction checks the TNP flag of the data transfer that it will process and automatically performs a 'waitForTransferComplete' if the TNP flag is not set.

Now consider the case where we don't want to overwrite a frame in the Receive buffer with new acquisition data until the frame has been processed; or perhaps we are acquiring sequential frames into a multi-frame Receive buffer, and we don't want the hardware acquisition to get too far ahead of the frame that we are processing. (If we were to let this

happen, our processed image would not be displayed in “real” time; in other words, the displayed image would lag behind the movement of the transducer.) For these situations, we can set the `'waitForProcessing'` condition in the `'transferToHost'` command with an argument that references the `'transferToHost'` data that we want to make sure is processed before allowing the hardware sequencer to proceed. When the DMA transfer of the current `'transferToHost'` command is initiated, the TNP flag of the referenced `'transferToHost'` is checked, and if set, the hardware sequencer sets a pause before the *next* `'transferToHost'` action. [Why the next `'transferToHost'` instead of the current one? Because the hardware sequencer interrupt that has initiated the current transfer has already occurred, and the hardware sequencer is continuing to run, processing new acquisition events. Since we don't want to interrupt acquisition events, which may have critical timing, the next opportunity to stop the hardware sequencer is at the completion of the acquisition of a new frame, before the transfer of the new frame begins.]

Let's look at an example with a Receive Buffer that has ten frames. If we don't want DMA transfers to get more than one frame ahead of the frame being processing, we set `'waitForProcessing'` conditions in each `'transferToHost'` command that reference the previous `'transferToHost'` command. In this case, when the hardware sequencer encounters the `'transferToHost'` command for frame two, for example, it checks the TNP flag for the frame one `'transferToHost'`, and if set, inserts a pause before the next `'transferToHost'`, which in this case is for frame three. When the processing of frame one is complete, we can reset its associated TNP flag, using a `'markTransferProcessed'` SeqControl command. This cancels the `'pause'` condition set before the transfer of frame three. If the hardware sequencer has already reached the `'pause'` condition, and is waiting to transfer frame three, the resetting of the TNP flag will release the hardware sequencer to start transferring frame three. The software can proceed to start processing frame two, since it has already been transferred.

If the above descriptions of the various synchronizing mechanisms proved difficult to follow, you can at least take comfort in the fact that most scripts can be designed so they don't need to use these mechanisms, and can be programmed to run entirely in asynchronous mode. (More on how this works in a later section.) The scripts that typically require synchronization mechanisms are scripts that run once, such as our current example, and scripts that include external trigger SeqControl actions.

8.2 Running the example script with the hardware.

Getting back to running our example script with the hardware, we turned off simulate mode above by setting `Resource.Parameters.simulateMode = 0`. This is the default condition, so we could have accomplished the same thing by simply deleting the line. But there is a potential problem with executing our example script. So far, our example script simply runs once and stops. In this case, the hardware and software sequencers each start at the first Event, and process Events until they reach the end. The software sequencer will ignore the transmit and receive actions in the first two events, as well as the `'transferToHost'` SeqControl command in the second Event, and proceed to perform the processing actions specified in the third Event. This situation is depicted below where the events executed by the hardware and software sequencer when running with the hardware are highlighted in bold text:

Hardware Sequencer

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;         % use 1st Rcv structure.
Event(1).recon = 0;       % no reconstruction.
Event(1).process = 0;     % no processing
Event(1).seqControl = 1;  % transfer data to host
    SeqControl(1).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;          % no TX structure.
Event(2).rcv = 0;         % no Rcv structure.
Event(2).recon = 0;       % no reconstruction.
Event(2).process = 1;     % call processing function
Event(2).seqControl = 0;
```

Software Sequencer

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;         % use 1st Rcv structure.
Event(1).recon = 0;       % no reconstruction.
Event(1).process = 0;     % no processing
Event(1).seqControl = 1;  % transfer data to host
    SeqControl(1).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;          % no TX structure.
Event(2).rcv = 0;         % no Rcv structure.
Event(2).recon = 0;       % no reconstruction.
Event(2).process = 1;     % call processing function
Event(2).seqControl = 0;
```

Looking at the above Events we can see that we have a potential problem. What if the hardware sequencer has not completed acquisition and transfer of the acquired data before the external processing starts? This will almost certainly be the case, since the software sequencer has nothing to do except call the external processing function. This would then lead to our external processing function processing an empty buffer or perhaps a partially transferred acquisition data set. One way to prevent this condition is by adding a 'waitForTransferComplete' SeqControl command to the second Event. The 'waitForTransferComplete' argument should be the number of the SeqControl structure that has the 'transferToHost' command we want to wait for, which in this case is SeqControl number one. When we execute a 'waitForTransferComplete' SeqControl action, we should also follow it with a 'markTransferProcessed' action, as we need to reset the TNP flag so we can wait for it to go high again with the next transfer. When both of these actions are in the same Event, the 'waitForTransferComplete' action is executed, then the process action of the event, and then the 'markTransferProcessed' action. [Note: If a recon action is specified in the event, the 'waitForTransferComplete' and 'markTransferProcessed' actions are automatically implemented and don't need to be specified by the user.]

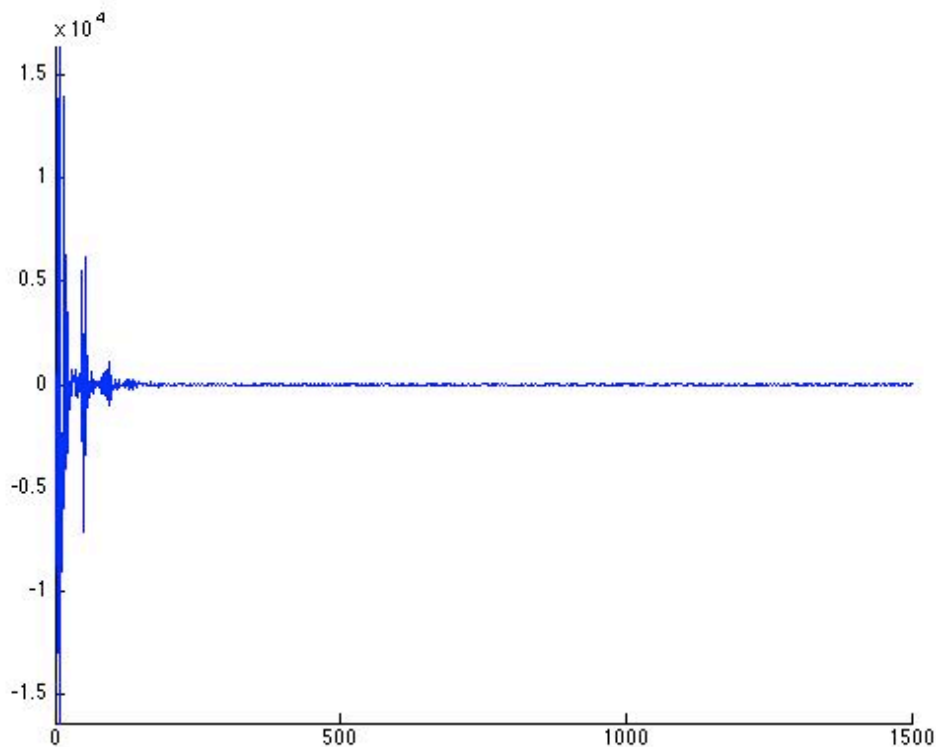
With the above modifications, our second Event now becomes:

```

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0; % no TX structure.
Event(2).rcv = 0; % no Rcv structure.
Event(2).recon = 0; % no reconstruction.
Event(2).process = 1; % call processing function
Event(2).seqControl = [2,3]; % wait for data to be transferred
    SeqControl(2).command = 'waitForTransferComplete';
    SeqControl(2).argument = 1;
    SeqControl(3).command = 'markTransferProcessed';
    SeqControl(3).argument = 1;

```

Now we can recompile and run our script with VSX, this time acquiring data with the VDAS hardware. If we have an L11-4v probe connected, but not applied to a medium, we will get a plot that looks something like the following.



When running a script, the transmit high voltage defaults to the minimum level of 1.6 volts (3.2 volts peak-to-peak) for safety reasons. For a script that runs only once, this can be a problem, but for this example, the minimum high voltage will suffice. The large amplitude signal at the start of the plot is the actual transmit burst, or rather the signal from the transmit burst that made it through the transmit protection circuitry in front of the TGC and subsequent analog processing prior to the A/D converter. When the `Receive.startDepth` is set to zero, the start of A/D sampling in the Verasonics system is same as the start of the transmit delay timers (which in this case are set to delays of 0), so the transmit burst is included in the receive period. Following the transmit burst, we see several echo returns at decreasing amplitudes. These echos are the reflections from the lens surface of transducer, which repeat as the sound reflects off the transducer surface and travels forward again.

9.0 Running a Sequence Continuously

While we can repeatedly run our example sequence by continually pressing the ‘freeze/unfreeze’ button in the GUI window, this is not very convenient. In general, we would like our sequence to repeat continuously, until we tell it to freeze or stop. This is easily accomplished by adding an Event with a ‘jump’ command at the end of our sequence to return to the beginning. We can modify our Event list as follows:

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1; % use 1st TX structure.
Event(1).rcv = 1; % use 1st Rcv structure.
Event(1).recon = 0; % no reconstruction.
Event(1).process = 0; % no processing
Event(1).seqControl = 1; % transfer data to host
SeqControl(1).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0; % no TX structure.
Event(2).rcv = 0; % no Rcv structure.
Event(2).recon = 0; % no reconstruction.
Event(2).process = 1; % call processing function
Event(2).seqControl = [2,3]; % wait for data to be transferred
SeqControl(2).command = 'waitForTransferComplete';
SeqControl(2).argument = 1;
SeqControl(3).command = 'markTransferProcessed';
SeqControl(3).argument = 1;

Event(3).info = 'Jump back to Event 1.';
Event(3).tx = 0; % no TX structure.
Event(3).rcv = 0; % no Rcv structure.
Event(3).recon = 0; % no reconstruction.
Event(3).process = 0; % no processing
Event(3).seqControl = 4; % jump back to Event 1.
SeqControl(4).command = 'jump';
SeqControl(4).argument = 1;
```

But wait! There are some problems with the above sequence of Events. The first problem is that there is no time allotted between acquisition events. The hardware sequencer will execute the acquisition actions in Events 1, and immediately jump back to repeat these actions again. The pulse repetition rate will be quite high, and could lead to problems with too much power being transmitted by our transducer. We need to insert some time between acquisitions to bring the pulse repetition frequency (PRF) down to something reasonable. To accomplish this, we can use the ‘timeToNextAcq’ SeqControl command. This command starts a timer at the start of the acquisition period in the Event that it is included, with the timer set to count down the number of microseconds specified in the SeqControl argument. When the next acquisition Event (an Event with either a TX reference, or both a TX and a Receive reference) occurs, if the timer has not expired, a pause state is entered until the timer expires. The start of transmit for the next acquisition event will occur immediately after the timer expires. If the timer expires before the next acquisition Event is reached in the sequence, a warning is printed out on the Matlab command line, to let the user know that the time duration specified could not be met. The ‘timeToNextAcq’ command, as its name implies, only works between successive acquisition Events, but is quite useful in setting precise PRFs. For simply inserting a time period between any two Events, the ‘noop’ (no operation) command can be used instead.

For our example sequence, we will acquire the acquisition data with a wait period of 50 msec between the acquisition Event (Event 1), giving 20 “frames” per second. This time period requires a new SeqControl structure, so we will re-order our SeqControl numbers as shown below. Note that we can have up to three SeqControl references in the same Event.

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;         % use 1st Rcv structure.
Event(1).recon = 0;       % no reconstruction.
Event(1).process = 0;     % no processing
Event(1).seqControl = [1,2];
    SeqControl(1).command = 'timeToNextAcq'; % set 50 msec PRF
    SeqControl(1).argument = 50000;
    SeqControl(2).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;          % no TX structure.
Event(2).rcv = 0;         % no Rcv structure.
Event(2).recon = 0;       % no reconstruction.
Event(2).process = 1;     % call processing function
Event(2).seqControl = [3,4]; % wait for data to be transferred
    SeqControl(3).command = 'waitForTransferComplete';
    SeqControl(3).argument = 2;
    SeqControl(4).command = 'markTransferProcessed';
    SeqControl(4).argument = 2;

Event(3).info = 'Jump back to Event 1.';
Event(3).tx = 0;          % no TX structure.
Event(3).rcv = 0;         % no Rcv structure.
Event(3).recon = 0;       % no reconstruction.
Event(3).process = 0;     % no processing
Event(3).seqControl = 5; % jump back to Event 1.
    SeqControl(5).command = 'jump';
    SeqControl(5).argument = 1;
```

With the above changes, we have fixed the problem of having too high a PRF rate; but there is still another potential problem. Since we have allocated only a single RcvBuffer frame, we are acquiring into the same frame that we are processing to extract a channel's RF data for display. If we don't allot sufficient time between acquisitions for processing, the hardware could be overwriting data in the RcvBuffer at the same time we are accessing it for display. To prevent this from happening, we need to use another of the hardware/software sequencer synchronizing mechanisms.

The easiest way to fix the possible overwriting of acquisition data is to place a 'sync' command in the same event as the process function. This will pause the hardware sequencer at this event until the software completes its processing and cancels the pause. Note that the software sequencer acts on the 'waitForTransferComplete' command in an event *before* any processing action and on the 'sync' command *after* any processing action.

With the 'sync' command synchronization, the hardware sequencer will perform the acquisition actions in Event 1, then pause in Event 2. The pause will be in effect until we release the hardware sequencer to run when the processing in Event 2 is complete.

Adding the 'sync' command to our sequence gives the following:

```
% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;         % use 1st Rcv structure.
Event(1).recon = 0;       % no reconstruction.
```

```

Event(1).process = 0;      % no processing
Event(1).seqControl = [1,2];
    SeqControl(1).command = 'timeToNextAcq';
    SeqControl(1).argument = 50000;
    SeqControl(2).command = 'transferToHost';

Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;          % no TX structure.
Event(2).rcv = 0;         % no Rcv structure.
Event(2).recon = 0;       % no reconstruction.
Event(2).process = 1;     % call processing function
Event(2).seqControl = [3,4,5]; % wait for data to be transferred
    SeqControl(3).command = 'waitForTransferComplete';
    SeqControl(3).argument = 2;
    SeqControl(4).command = 'markTransferProcessed';
    SeqControl(4).argument = 2;
    SeqControl(5).command = 'sync';

Event(3).info = 'Jump back to Event 1.';
Event(3).tx = 0;          % no TX structure.
Event(3).rcv = 0;         % no Rcv structure.
Event(3).recon = 0;       % no reconstruction.
Event(3).process = 0;     % no processing
Event(3).seqControl = 6; % jump back to Event 1
    SeqControl(6).command = 'jump';
    SeqControl(6).argument = 1;

```

With the 'sync' command in our process event, the 'timeToNextAcq' command in the Event 1 is not really needed, as identified by the shaded lines above. Removing this command will let the acquisition run as fast as the processing will allow. Of course, if we want to specify precisely the PRF at which acquisition repeats, we can leave the 'timeToNextAcq' command in the Event, but in this case, we must make sure that the specified time to wait is longer than the processing time; otherwise, we will get a warning message on the Matlab command line that our 'timeToNextAcq' wait time is too short, along with the amount of time we would need to add to make our script meet timing. [It is not unusual to get one or two warning messages as the script starts up, due to initialization processes.]

There are a few other things to note in the revised script. As mentioned previously, in the 'Call external Processing function' Event, note that it is permitted to place the 'waitForTransferComplete', the 'markTransferProcessed' and 'sync' SeqControl commands in the same Event with the processing action. The 'waitForTransferComplete' command will be executed before processing and the 'markTransferProcessed' and 'sync' command after.

By jumping back to the first Event from the last, we have put our sequence into an infinite loop. It should be noted that while our sequence is executing, Matlab operation is suspended, since the 'runAcq.c' executing function has not returned. This would prevent us from exiting our script by clicking on the GUI window close button, or processing any GUI controls (GUI controls will appear to change, but their callback functions will not execute). Consequently, we always need a 'returnToMatlab' SeqControl command somewhere in our sequence, so that we can allow Matlab to process any pending GUI actions. By default, a 'jump' SeqControl command back to Event 1 also implements an automatic 'returnToMatlab' SeqControl command. This is to insure that there is at least one 'returnToMatlab' command in a repeating script.

Running our sequence with the above changes, we see a continuously updating plot of the receive data from channel 32. We can now use the HighVoltage slider to increase the

transmit high voltage to something reasonable, such as 35 volts. Placing our L11-4v transducer on a CIRS 040GSE phantom over a vertical row of targets, we get the following plot.

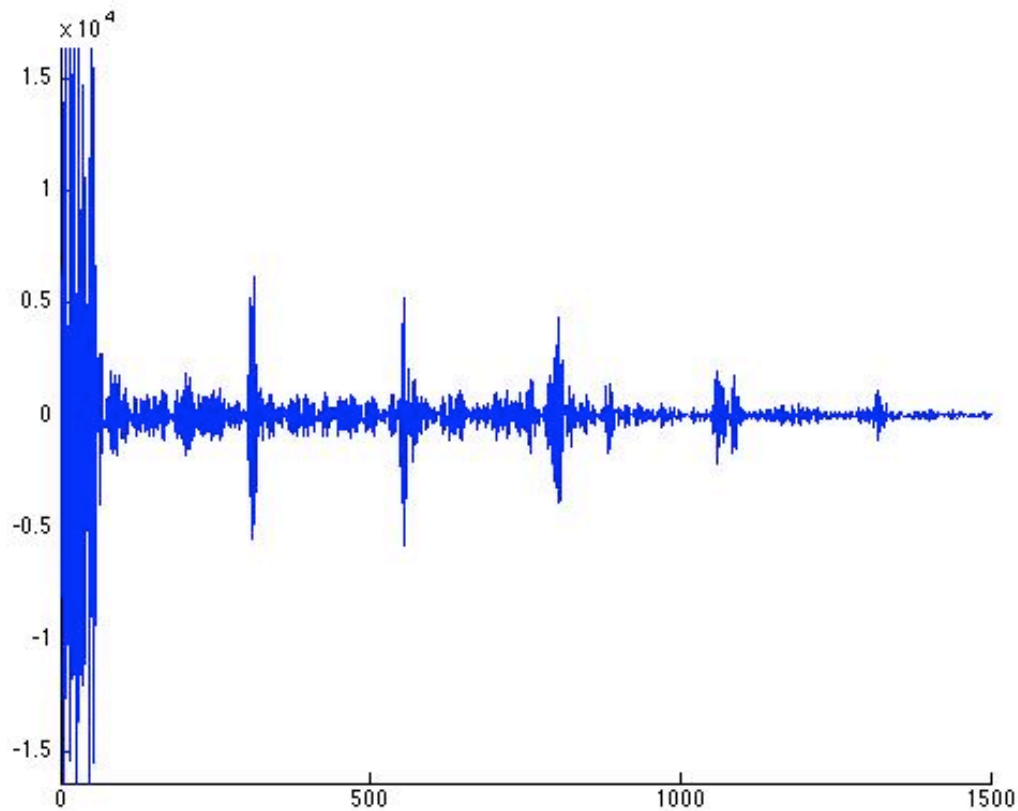


Fig. 9.1 Example script plot of channel 32 with L7-4 transducer positioned on RMI404GS phantom.

The multiple targets seen in Fig. 9.1 are from the wire targets in the phantom, and since we are looking at the signal from a single element, we see all the targets in the element's field of view.

10.0 Adding a GUI Control

Our example script displays a single channel's RF data for the single acquisition. When we constructed our external processing function, we had it try to get the number of the channel to display from a variable called 'myPlotChnl' in the base Matlab workspace.

```
if evalin('base','exist('myPlotChnl','var'))
    channel = evalin('base','myPlotChnl');
else
    channel = 32; % Channel no. to plot
end
```

If this variable doesn't exist, we default to plotting the RF data for channel 32. So all we need to do to plot a different channel's data is to set the myPlotChnl variable in the base workspace to the desired channel number. Since we can't execute Matlab statements on the command line while our script is running, we need to construct a GUI control to set the variable. GUI controls in the Matlab environment execute in their own thread, and can schedule a callback function that is executed when the our script performs a 'returnToMatlab'. [While the Matlab GUI controls can change state while our sequence is running, the callbacks associated with the controls are only put into Matlab's event queue, to be executed when control is returned to the main Matlab execution thread.]

In the Verasonics software, we have provided a way to specify user GUI controls and callback functions within the SetUp scripts, thus allowing the controls to be kept with the script that uses them. This mechanism incidentally also provides a way to execute Matlab commands just before our script runs, which can be useful for scripts that need to initialize something just before running. The object that the user script specifies is the UI object and has the following form:

```
UI =
    Statement    string        String that can be evaluated as Matlab command.
    Control      cell array    Attribute/value pairs for UIControl statement.
    Callback     cell array    Cell array of strings for Matlab function.
```

The `Statement` and/or the `Control` attribute should be provided, but unused attributes can be missing or empty. The `Statement` attribute is a string that represents a Matlab command that can be executed with the Matlab 'eval' function. This statement will be executed by VSX just before our script is loaded and run. It is sometimes needed to build more complex UI controls, or to just initialize some parameter. Next, the `Control` cell array holds the input parameters (mainly attribute/value pairs) for the Matlab 'uicontrol' command. This command is used to add UI objects to the GUI window, and is called by VSX prior to loading and running a script. Finally, we have the `Callback` attribute, which is a cell array of strings that compose a Matlab callback function. This function will be created by VSX in the /tmp directory of the system and the path to the /tmp directory will be added to the Matlab path. The first line must be the prototype of the function to be created, whose name should be referenced as the callback function for a `UI.Control` attribute.

An example will make this UI creation mechanism easier to understand. Let's add a UI control to set the channel that we want to plot in our example script. There are several ways this could be implemented, but we choose to use a slider control, with a text field underneath that indicates the channel number. In this case, we don't need the `Statement` attribute in our UI object.

To define the attributes needed for a slider UI object, we can use the following definition.

```
% - Create UI control for channel selection
nr = Resource.Parameters.numRcvChannels;
UI(1).Control = {'UserB1','Style','VsSlider',...
    'Label','Plot Channel',...
    'SliderMinMaxVal',[1,64,32],...
    'SliderStep',[1/nr,8/nr],...
    'ValueFormat','%3.0f'};
```

Verasonics has provided several built-in UI control functions, which are documented in the Sequence Programming Manual. In this case, we are asking for a control of style 'VsSlider', which is actual three UI controls in a group - a text label, a slider, and an edit box linked to the slider control. Moreover, there are a number of predefined locations for Verasonics custom UI controls, as shown in Fig. 10.1 below.

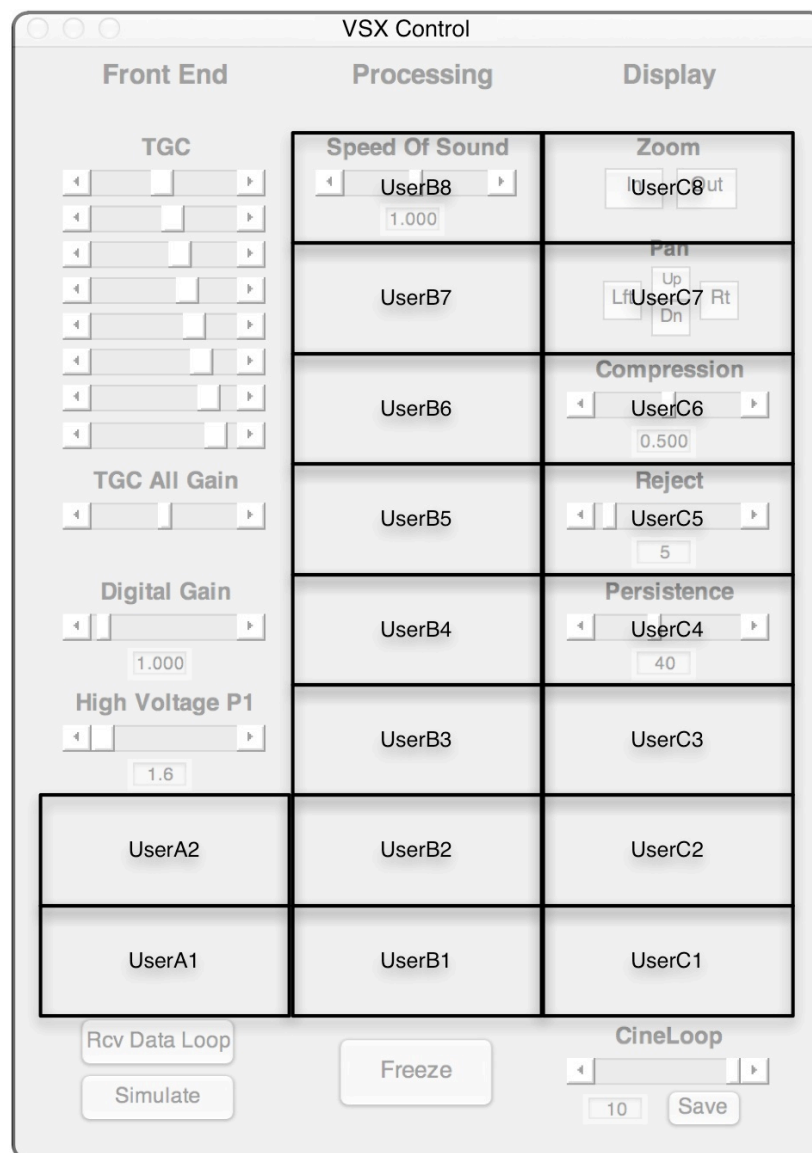


Fig. 10.1 Locations available for Verasonics custom UI controls

Our `UI(1).Control` definition is asking for the control to be created in the UserB1 location of the GUI under the processing column. The 'Label' attribute is the name we want for the control, and the 'SliderMinMaxValue' is an array that defines the slider minimum value, the maximum value, and the initial value. The 'SliderStep' attribute defines the size of the step when one clicks the slider left/right buttons, and the step when one clicks in the area left or right of the slider control. Finally, the 'ValueFormat' attribute is the format for the value to be placed in the edit box, in this case a floating point number with 3 digits.

We next define the callback for our `UI(1).Control`, which is the code that will get executed when the slider or edit box value is changed.

```
UI(1).Callback = {'assignin('base','myPlotChnl',round(UIValue))'};
```

In this case, our callback code consists of just one line, since the VSX software automatically adds a preamble to the callback that manages the slider and edit box linkage, and the extraction of the value set by the user. This value is available to our user code in the variable 'UIValue'. Our one line statement simply puts the value of the control in the variable 'myPlotChnl' in the base workspace, where it can be read by our external function.

In the above example, the callback code is very short, and easy to encode as quoted text for the cell array definition. For longer code sections, this encoding can be tedious, and difficult to read or debug. For these cases, a utility function, named 'text2cell' is provided which will automatically perform the cell array encoding for a section of code. To use the 'text2cell' function, write your callback or external function code at the end of your setup script, after placing a return at the end of the SetUp script definitions, so the added code won't get executed. (Matlab may warn you in the editor that the first statement of the added code can't be reached.) Bracket each added callback or external function with a unique identifier on its own line, which should start with a comment character, '%' and followed immediately by the delimiter characters. For example, the `UI(1).Callback` statement above could be written as follows:

```
return % Place this return at end of script to prevent executing code below
%CB#1
assignin('base','myPlotChnl',round(UIValue));
%CB#1
```

Then in the definition of the `UI(1).Callback`, use the following statement:

```
> UI(1).Callback = text2cell('%CB#1');
```

For encoding an external function, which will also be created in the tmp directory, one can use the following command (note different unique delimiter):

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

In the above callback definition and function definition, the only input to the text2cell utility function is the delimiter, with the function searching the calling script file by default. If a different file is to be searched for the text to encode, a filename can be included as the first input parameter, before the delimiter string. The text2cell function will scan the script lines of code, looking for the specified delimiter, then start encoding the lines of code up to the

next instance of the delimiter. Be sure to chose a different delimiter to bracket each callback or external function provided.

Hints for debugging dynamically created callback or external functions: If we run our script with VSX and our callback or external function fails to operate properly, add a 'keyboard' statement in the callback or external function code in the SetUp script at the point where one wants to break execution. Compile and run the SetUp script again using VSX, and then exercise the callback or external function. Matlab will stop at the keyboard command and return control to the command window. At this point, we can open the callback or external function in the Matlab editor using the command:

```
>edit UserXXCallback.m
```

where XX is the location code (B1 in this example) for a Verasonics callback, or for an external function

```
>edit myExtFuncName.m
```

The function will open in the editor, with the point where execution stopped shown with an error. At this point, one can use the debug controls in the editor window to single step through the function and observe execution. To make changes to the function, quit the script and make code changes in the SetUp script definition, recompile and run again. When the function has been debugged, the keyboard command can be removed and the script recompiled.

Running our script with the added UI control, we see a new slider control in the lower central portion of the GUI window with a label and text edit box underneath. We can set the channel number to plot with either the slider or by typing in the channel number in the text box.

The entire script with UI control functions added is listed below for reference. Note that we have also coded our external function at the end of the script and encoded it into the cell array `EF(1).Function`. This array will be read by VSX at load time and the external function dynamically created in the tmp directory. Including this code eliminates the need for a separate external function file and allows our single Setup script file to contain all the resources necessary for running. (Be sure to delete the myProcFunction.m file from the main directory, or it will be used instead of the function in the tmp directory, as it is found first when Matlab searches for the file to execute.)

The text below can be copied and pasted into the Matlab editor to save and run.


```

% File name SetUpL11_4vAcquireRF.m
% - Synchronous acquisition into a single RcvBuffer frame.

clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128; % no. of transmit channels
Resource.Parameters.numRcvChannels = 128; % change to 64 for Vantage 64 system
Resource.Parameters.connector = 1; % trans. connector to use (V 256).
Resource.Parameters.speedOfSound = 1540; % speed of sound in m/sec
Resource.Parameters.fakeScanhead = 1; % optional (if no L7-4)
Resource.Parameters.simulateMode = 0; % runs script with hardware

% Specify media points
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L11-4v transducer is 'known' transducer.

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048;
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 1; % minimum size is 1 frame.

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [5.208,0.67,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
TGC(1).CntrlPts = [500,590,650,710,770,830,890,950];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify Receive structure array.
Receive(1).Apod = ones(1,128); % if 64ch Vantage, = [ones(1,64) zeros(1,64)];
Receive(1).startDepth = 0;
Receive(1).endDepth = 200;
Receive(1).TGC = 1; % Use the first TGC waveform defined above
Receive(1).mode = 0;
Receive(1).bufnum = 1;
Receive(1).framenum = 1;
Receive(1).acqNum = 1;
Receive(1).sampleMode = 'NS200BW';
Receive(1).LowPassCoef = [];
Receive(1).InputFilter = [];

% Specify an external processing event.
Process(1).classname = 'External';
Process(1).method = 'myProcFunction';
Process(1).Parameters = {'srcbuffer','receive',... % name of buffer to process.
                        'srcbufnum',1,...
                        'srcframenum',1,...
                        'dstbuffer','none'};

% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1; % use 1st TX structure.
Event(1).rcv = 1; % use 1st Rcv structure.
Event(1).recon = 0; % no reconstruction.

```



```

Event(1).process = 0;      % no processing
Event(1).seqControl = [1,2];
    SeqControl(1).command = 'timeToNextAcq';
    SeqControl(1).argument = 50000;
    SeqControl(2).command = 'transferToHost';
Event(2).info = 'Call external Processing function.';
Event(2).tx = 0;          % no TX structure.
Event(2).rcv = 0;         % no Rcv structure.
Event(2).recon = 0;       % no reconstruction.
Event(2).process = 1;     % call processing function
Event(2).seqControl = [3,4,5]; % wait for data to be transferred
    SeqControl(3).command = 'waitForTransferComplete';
    SeqControl(3).argument = 2;
    SeqControl(4).command = 'markTransferProcessed';
    SeqControl(4).argument = 2;
    SeqControl(5).command = 'sync';
Event(3).info = 'Jump back to Event 1.';
Event(3).tx = 0;          % no TX structure.
Event(3).rcv = 0;         % no Rcv structure.
Event(3).recon = 0;       % no reconstruction.
Event(3).process = 0;     % no processing
Event(3).seqControl = 6; % jump back to Event 1
    SeqControl(6).command = 'jump';
    SeqControl(6).argument = 1;

% Create UI control for channel selection
nr = Resource.Parameters.numRcvChannels;
UI(1).Control = {'UserB1','Style','VsSlider',...
    'Label','Plot Channel',...
    'SliderMinMaxVal',[1,128,32],...
    'SliderStep', [1/nr,8/nr],...
    'ValueFormat', '%3.0f'};
UI(1).Callback = text2cell('%CB#1');
% Create External function for plotting channel data
EF(1).Function = text2cell('%EF#1');

% Save all the structures to a .mat file.
save('L11-4vAcquireRF');
return % Place this return to prevent executing code below

%CB#1
assignin('base','myPlotChnl', round(UIValue));
%CB#1

%EF#1
myProcFunction(RData)
persistent myHandle
% If 'myPlotChnl' exists, read it for the channel to plot.
if evalin('base','exist('myPlotChnl','var'))
    channel = evalin('base','myPlotChnl');
else
    channel = 32; % Channel no. to plot
end
% Create the figure if it doesn't exist.
if isempty(myHandle) || ~ishandle(myHandle)
    figure;
    myHandle = axes('XLim',[0,1500],'YLim',[-16384 16384], ...
        'NextPlot','replacechildren');
end
% Plot the RF data.
plot(myHandle,RData(:,channel));
drawnow
%EF#1

```

11.0 Designing an Asynchronous Script

Our example script to acquire RF data had to be programmed to run synchronously, since we only allocated a single RcvBuffer frame to hold our acquired data. The hardware sequencer was required to pause before acquiring new acquisition data, if the software had not yet completed processing of the frame. Conversely, if the DMA transfer was still in progress, the software sequencer was required to 'waitForTransferComplete' before processing the acquisition. Moreover, these actions are all occurring in series, and our script is not taking advantage of parallel operations. While this synchronous operation is fine for our simple example script, and synchronizing mechanisms are sometimes required for proper sequence function, there are certain drawbacks to synchronous operation.

First of all, most synchronous scripts are paced by the rate of software processing, which is not a fixed parameter. In modern operating systems, there are typically many background tasks which demand processor resources, and these tasks can cause variations in the rate of processing. We can compensate for this variability with 'timeToNextAcq' commands, but if we set the 'timeToNextAcq' time periods too close to the typical processing time, we will get warning messages when an occasional long processing period occurs. These occasional long processing times are particularly obnoxious when it comes to scripts that acquire Doppler data. With Doppler acquisitions, any tiny perturbation in the PRF will lead to glitches and pops in the Doppler spectrum and audio.

Secondly, synchronous scripts are harder to write, since we have to deal with the various SeqControl mechanisms involved. This difficulty gets worse in longer, more complex sequences that deal with many frames of RF, and possible mode changes. For example, if we get an index wrong in a 'waitForTransferComplete' command, we can hang up both the software and hardware sequencers. [There are timeouts that limit how long a command will wait before giving up. These are described in the Sequence Programming Manual.]

Thirdly, it is often desired to capture acquisition frames at a rate much higher than processing would allow. For example, we might want to capture a brief time period of images at a very high rate following some event, such as when a shear wave is generated. Here we may want to capture frames at rates of several thousand frames per second, and process all the data afterwards. Or possibly we just want to always capture acquisition data frames at whatever rate the speed of sound (or the DMA transfer limit) allows, and continuously store these data in a circular buffer. For processing and display, we might want to run as fast as our computing power allows, processing the most recently acquired frame in the buffer when the processing of a new frame begins. For this type of processing, we need to design an asynchronous sequence.

We will now proceed to modify our example script for acquiring RF data so that it executes asynchronously. The first steps in designing an asynchronous sequence are 1) deciding on acquisition frame rates and 2) choosing the number of RcvBuffer frames to allocate. Let's choose a frame rate (or in our case, an acquisition rate) of 100 frames per second. Since we only have a single acquisition in our 'frame', we can easily set the acquisition rate using a 'timeToNextAcq' SeqControl command. If we want the acquisitions to repeat at 100 per second, we will need a 'timeToNextAcq' period of 10 msec (100 fps -> 10 msec period).

```

% Specify sequence events.
Event(1).info = 'Acquire RF Data.';
Event(1).tx = 1;           % use 1st TX structure.
Event(1).rcv = 1;          % use 1st Rcv structure.
Event(1).recon = 0;        % no reconstruction.
Event(1).process = 0;      % no processing
Event(1).seqControl = 1;   % wait 10000 usec
    SeqControl(1).command = 'timeToNextAcq';
    SeqControl(1).argument = 10000;

```

The next step is choosing how many frames to allocate for our RcvBuffer. The idea here is that when our sequence is run, acquisition will march through the frames of the RcvBuffer, and wrap back to the start when the last frame in the buffer is acquired. To prevent overwriting a frame that is being processed, the time to acquire all the frames in the buffer must be longer than the processing time. In our example case, the processing is fairly simple, and is basically the time to call the plot function in Matlab. If we want to be precise, we could measure the processing time using Matlab's 'tic' and 'toc' functions, but let's just guess that it takes 20 msec to do a plot (the actual time will depend on the speed of the computer). If it takes 20 msec to make a plot, we will then need more than two frames in our RcvBuffer to be safe (at 10 msec per frame). Let's be super safe and allocate 10 frames for our RcvBuffer. This will allow processing to take up to 100 msec to complete without the possibility of overwriting the RF data being processed.

```

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048;
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 10;

```

Now we need to modify the objects in our script to use all of the 10 frames in the RcvBuffer. The TW and TX and TGC definitions can be re-used for each frame, so we don't need any addition to these structures. For Receive, however, we need a separate structure for each acquisition that goes to a unique location in the RcvBuffer. Remember that VSX will be appending the storage location in local memory to each Receive structure, and these must each be to different addresses. Moreover, each of the Resource.RcvBuffer.numFrames acquisition frames must be transferred to a different address in the host computer buffer. The easiest way to define the Receive structures for multiple frames is to first define the general form of the Receive structure in a Matlab 'struct' definition. This definition can then be replicated for the number of Receive structures needed in our script. Next we can write a 'for' loop that sets the unique values for each Receive structure for all the frames. The new definition for the Receive structures is shown below.

```

% Specify Receive structure array -
Receive = repmat(struct(...
    'Apod', ones(1,128), ... % if 64ch, = [ones(1,64) zeros(1,64)];
    'startDepth', 0, ...
    'endDepth', 200, ...
    'TGC', 1, ...
    'mode', 0, ...
    'bufnum', 1, ...
    'framenum', 1, ...
    'acqNum', 1, ...
    'sampleMode', 'NS200BW', ...
    'LowPassCoef', [], ...
    'InputFilter', [], ...
    1, Resource.RcvBuffer(1).numFrames);

% - Set event specific Receive attributes.
for i = 1:Resource.RcvBuffer(1).numFrames
    Receive(i).framenum = i;
end

```

In the above code we have use the Matlab ‘repmat’ command to replicate our Receive generic definition multiple times. The ‘for’ loop below the structure definition sets the acquisition frame specific attributes, which in this case is the `framenum` attribute.

Now we come to the Process definition, which determines the source buffer type, number and frame number to send to our external function. Without any change in this structure, we will always process frame 1 of the RcvBuffer, since that is what is specified for the `srcframenum` value. The frame that we would like to process is the most recent frame transferred to the RcvBuffer, as this will provide us with the most current data and the longest processing time before the acquisition overwrites the same frame again. We can indicate that we want the most recent frame transferred by entering a -1 for the frame number. In this case, the software will determine the last frame transferred to the RcvBuffer specified, and provide it as input to the external processing function.

```

% Specify an external processing event.
Process(1).classname = 'External';
Process(1).method = 'myProcFunction';
Process(1).Parameters = {'srcbuffer', 'receive', ... % name of buffer to process.
    'srcbufnum', 1, ...
    'srcframenum', -1, ... % process the most recent frame.
    'dstbuffer', 'none'};

```

The last structures to modify in our example script are the Event structures. Instead of specifying the Event structure numbers directly, we will use a variable `n`, that we will increment by one after each new Event structure defined. This has the advantage that we can insert or delete Events in the middle of our sequence without changing all the Event indices after the addition or deletion. We will use a ‘for’ loop to define the multiple acquisitions for all of the frames in the RcvBuffer.

At the beginning of the Event structure definitions, we will define the SeqControl numbers that can be reused in the sequence, since there is no point in defining them multiple times in the ‘for’ loop of Events. We then use the variable, `nsc`, to define the index of the next SeqControl structure that we will define dynamically in our ‘for’ loop. In this case, the SeqControl command that must be defined uniquely for each frame is the ‘transferToHost’ command, which must be repeated for each frame. VSX will replace each ‘transferToHost’ command with a ‘DMA’ command, and that ‘DMA’ command must transfer the last acquisition frame’s data from its unique location in local memory to a unique location in the

RcvBuffer. We therefore need a unique 'transferToHost' SeqControl command for every frame in our sequence. The 'transferToHost' commands will get their source and destination information from the unique Receive structure(s) used for the frame.

Our new Event list is as follows:

```
% Specify sequence events.
SeqControl(1).command = 'timeToNextAcq';
SeqControl(1).argument = 10000;
SeqControl(2).command = 'jump';
SeqControl(2).argument = 1;
nsc = 3; % start index for new SeqControl

n = 1; % start index for Events
for i = 1:Resource.RcvBuffer(1).numFrames
    Event(n).info = 'Acquire RF Data.';
    Event(n).tx = 1; % use 1st TX structure.
    Event(n).rcv = i; % use unique Receive for each frame.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 0; % no processing
    Event(n).seqControl = [1,nsc]; % set TTNA time and transfer
        SeqControl(nsc).command = 'transferToHost';
        nsc = nsc + 1;
    n = n+1;

    Event(n).info = 'Call external Processing function.';
    Event(n).tx = 0; % no TX structure.
    Event(n).rcv = 0; % no Rcv structure.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 1; % call processing function
    Event(n).seqControl = 0;
    n = n+1;
end
Event(n).info = 'Jump back to Event 1.';
Event(n).tx = 0; % no TX structure.
Event(n).rcv = 0; % no Rcv structure.
Event(n).recon = 0; % no reconstruction.
Event(n).process = 0; % no processing
Event(n).seqControl = 2; % jump back to Event 1.
```

Note also that we have eliminated the 'waitForTransferComplete', 'markTransferProcessed' and 'sync' SeqControl commands in our 'Call external processing function' Event. The reason these are no longer needed is that by specifying the most recently transferred frame to process, we know that 1) the frame is available for processing, and 2) that there will be an amount of time equal to $[(\text{Resource.RcvBuffer.numFrames} - 1) * (\text{acquisition frame period})]$ before acquisition returns to overwrite the data in our frame.

This completes the changes needed to convert our synchronous example sequence to an asynchronous one. In this new sequence, acquisition will run at precisely 100 frames per second, without ever stopping until we exit our script or press the 'freeze' UI button. Our processing to display the RF data from an individual channel will run as fast as our computer system will allow, providing a realtime display of the most recently acquired data. In this case, our display frame rate may be slightly variable, although this is usually not possible to detect at reasonable processing rates. The entire asynchronous script is reproduced below for reference.

```

% File name SetUpL11_4vAcquireRF.m
% - Asynchronous acquisition into multiple RcvBuffer frames.
clear all

% Specify system parameters
Resource.Parameters.numTransmit = 128;           % no. of transmit channels
Resource.Parameters.numRcvChannels = 128;        % change to 64 for Vantage 64 system
Resource.Parameters.connector = 1;               % trans. connector to use (V 256).
Resource.Parameters.speedOfSound = 1540;         % speed of sound in m/sec
Resource.Parameters.fakeScanhead = 1;            % optional (if no L7-4)
Resource.Parameters.simulateMode = 0;            % runs script with hardware

% Specify media points
Media.MP(1,:) = [0,0,100,1.0]; % [x, y, z, reflectivity]

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L7-4 transducer is 'known' transducer.

% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 2048;
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 10;

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [18,17,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
TGC(1).CntrlPts = [500,590,650,710,770,830,890,950];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify Receive structure array -
Receive = repmat(struct(...
    'Apod', ones(1,128), ... % if 64ch, = [ones(1,64) zeros(1,64)];
    'startDepth', 0, ...
    'endDepth', 200, ...
    'TGC', 1, ...
    'mode', 0, ...
    'bufnum', 1, ...
    'framenum', 1, ...
    'acqNum', 1, ...
    'sampleMode', 'NS200BW', ...
    'LowPassCoef', [],...
    'InputFilter', []),...
    1,Resource.RcvBuffer(1).numFrames);

% - Set event specific Receive attributes.
for i = 1:Resource.RcvBuffer(1).numFrames
    Receive(i).framenum = i;
end

% Specify an external processing event.
Process(1).classname = 'External';
Process(1).method = 'myProcFunction';
Process(1).Parameters = {'srcbuffer','receive',... % name of buffer to process.
    'srcbufnum',1,...
    'srcframenum',-1,... % process the most recent frame.
    'dstbuffer','none'};

```

```

% Specify sequence events.
SeqControl(1).command = 'timeToNextAcq';
SeqControl(1).argument = 10000;
SeqControl(2).command = 'jump';
SeqControl(2).argument = 1;
nsc = 3; % start index for new SeqControl

n = 1; % start index for Events
for i = 1:Resource.RcvBuffer(1).numFrames
    Event(n).info = 'Acquire RF Data.';
    Event(n).tx = 1; % use 1st TX structure.
    Event(n).rcv = i; % use unique Receive for each frame.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 0; % no processing
    Event(n).seqControl = [1,nsc]; % set TTNA time and transfer
        SeqControl(nsc).command = 'transferToHost';
        nsc = nsc + 1;
    n = n+1;

    Event(n).info = 'Call external Processing function.';
    Event(n).tx = 0; % no TX structure.
    Event(n).rcv = 0; % no Rcv structure.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 1; % call processing function
    Event(n).seqControl = 0;
    n = n+1;
end
Event(n).info = 'Jump back to Event 1.';
Event(n).tx = 0; % no TX structure.
Event(n).rcv = 0; % no Rcv structure.
Event(n).recon = 0; % no reconstruction.
Event(n).process = 0; % no processing
Event(n).seqControl = 2; % jump back to Event 1.

% - Create UI controls for channel selection
nr = Resource.Parameters.numRcvChannels;
UI(1).Control = {'UserB1','Style','VsSlider',...
    'Label','Plot Channel',...
    'SliderMinMaxVal',[1,128,32],...
    'SliderStep',[1/nr,8/nr],...
    'ValueFormat','%3.0f'};
UI(1).Callback = text2cell('%CB#1');
EF(1).Function = text2cell('%EF#1');

% Save all the structures to a .mat file.
save('L11-4vAcquireRF');
return % Place this return to prevent executing code below

%CB#1
assignin('base','myPlotChnl',round(UIValue));
%CB#1

%EF#1
myProcFunction(RData)
persistent myHandle
% If 'myPlotChnl' exists, read it for the channel to plot.
if evalin('base','exist('myPlotChnl','var')')
    channel = evalin('base','myPlotChnl');
else
    channel = 32; % Channel no. to plot
end
% Create the figure if it doesn't exist.
if isempty(myHandle) || ~ishandle(myHandle)
    figure;

```

```

myHandle = axes('XLim',[0,1500],'YLim',[-16384 16384], ...
               'NextPlot','replacechildren');
end
% Plot the RF data.
plot(myHandle,RData(:,channel));
drawnow
%EF#1

```

Note that the asynchronous sequence script is essentially the same code size as the synchronous script, yet has the advantage of supporting a RF cineloop capability. This can be demonstrated by running the program, scanning a target, and pressing the 'RcvDataLoop' toggle button on the GUI. When this button is pressed, the acquisition Events in the script are disabled, and processing will process the previously acquired frames in the RcvBuffer continuously. This is a convenient way of capturing data to test out various processing methods. 'RcvDataLoop' mode is essentially the same as specifying `Resource.Parameters.simulateMode = 2` at the start of our script.

Let's say we want to capture some RF data, and then experiment with various processing schemes. We can run our script with `Resource.Parameters.simulateMode = 0`, capturing RF data in our RcvBuffer and using some default processing mechanism. We can examine the captured data by switching to 'RcvDataLoop' mode after freezing the acquisition. Unfreezing the system will then start processing the data in the RcvBuffer from the oldest frame to the newest, repeating continuously. When we are satisfied with the RF data set captured, we exit our script, and save the RcvBuffer data (RcvData) to a file, using the Matlab command:

```
>save('MyRcvData.mat','RcvData')
```

If we want to know the number of last frame in the RcvBuffer that was acquired, we should also note or save the `Resource.RcvBuffer.lastFrame` value. This attribute is added to the `Resource.RcvBuffer` structure and set to the most recently acquired frame number when a script freezes or exits.

To work with our saved data at a later time, we can recompile our setup script with `Resource.Parameters.simulateMode = 2`. Before running the script with VSX, we load our saved RcvData with the Matlab command:

```
>load('MyRcvData.mat')
```

This will restore the RcvData cell array in the Matlab workspace. When we next run our script, VSX will see that there is already a RcvData array in the workspace. Provided the dimensions of the array found match the dimensions that is being called for in the script, VSX will use the found array instead of allocating a new one. Our script will then run in simulate mode 2 and process the loaded RF data as if it came from the Vantage hardware.

12.0 Performing Image Reconstruction

In our example script that we have developed so far, the main objective has been RF data acquisition, and our minimal processing function simply plots the acquired data for a channel. But since we have used an unfocused plane wave for the transmit waveform, even the simple acquisition of this script can be used to produce an image, although the spatial and contrast resolution will be reduced compared with a traditional scan that uses multiple ray lines. Image reconstruction in the Verasonics Research System is pixel-oriented, meaning that image reconstruction takes place only on predefined points in the field of the transducer. These predefined points are typically an array of points with uniform spacing in the x and z direction, similar to the pixel points in a photographic image. There are no intermediate data sets between the RF data and the reconstructed image points; in other words, there are no “beams” in this beamforming process.

This new method of image reconstruction has a number of advantages over more conventional beamforming methods. First of all, it processes only the RF data that contribute to each individual pixel, and this typically reduces the number of operations needed to form an image by almost an order of magnitude, making software “beamforming” in real-time possible. Secondly, it eliminates the need for scan conversion, since there are no intermediate data sets that need coordinate system conversion and/or interpolation. Thirdly, it supports new methods of image formation, such as imaging with unfocused transmit wavefronts, including full synthetic aperture imaging with individual element transmits. Finally, since the processing of each pixel is an independent process, the pixel array can be broken up into multiple regions that can be processed simultaneously by multi-core processing engines (CPUs and GPUs). This feature makes pixel-oriented image reconstruction particularly well suited for modern computing systems.

12.1 Defining the coordinate system with respect to the probe.

To add image reconstruction to our example script requires defining several new objects in our script. These new objects will define the spatial location of our reconstruction pixel grid relative to the transducer, and the type and number of reconstruction operations to be performed on each pixel. To understand the first of these elements, we need to define a coordinate system relative to the transducer. For a linear array, we define the coordinates as in figure 12.1 below. For a linear array, the position of a transducer element in the coordinate system is defined with respect to the center of the element. Defined as such, a linear array element’s position is determined solely by its x coordinate value, since y and z are both zero. The plane in which we will form an image is the x, z plane, where as in a traditionally presented ultrasound image, x represents the lateral direction, with the x axis at the top of the image and the negative x direction on the left. The z direction will be the axial, or depth direction, with the positive z direction pointing down.

We will be defining most distance measurements in wavelengths of the center frequency of the probe. Since wavelengths take into account the speed of sound in the media, defining length in wavelengths eliminates the need to know the speed of sound value in our reconstruction calculations. Also, since our sampling rate and pixel distances are defined in numbers per wavelength, and the maximum depth of penetration of ultrasound is roughly constant when expressed in wavelengths, our reconstruction processing in terms of

operations per pixel region will be essentially independent of transducer center frequency. This is very important for software beamforming, and allows us to have the same frame rate performance at a 10MHz center frequency as we have at 1 MHz.

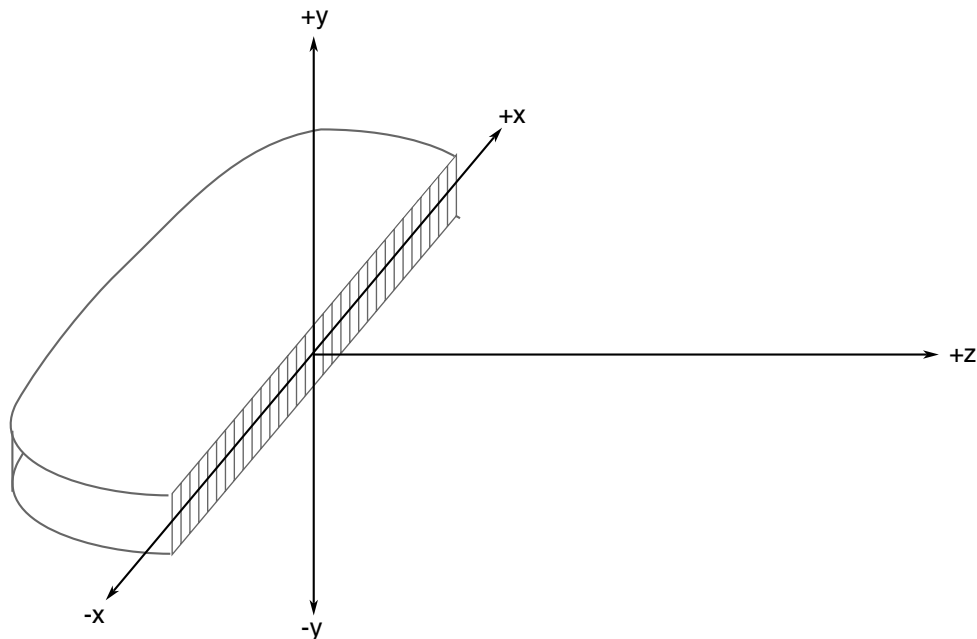


Fig. 12.1 Linear array transducer coordinate system.

12.2 The PData object definition

The PData object (short for Pixel Data) specifies the location of our pixel data array. For 2D scans and slices of 3D scans, the pixel data array always represents a rectangular area at some fixed location in our transducer coordinate system. As such, it has rows and columns of pixels where reconstruction can take place. For a linear array, the PData pixel array is typically defined to be the same width as the transducer with a height equal to the scan depth required, but this need not be the case. The PData region can be larger or smaller than the typical size, and doesn't have to start at the transducer face. Let's look at a typical PData definition for our example script.

```
% Specify PData structure array.
PData.PDelta = [Trans.spacing,0,0.5]; % x, y and z pixel deltas
startDepth = 5;
endDepth = 200;
PData.Size(1) = ceil((endDepth-startDepth)/PData.PDelta(3));
PData.Size(2) = ceil(128*Trans.spacing/PData.PDelta(1));
PData.Size(3) = 1; % 2D image plane
% PData.Origin is the location [x,y,z] of the upper left corner of the array.
PData.Origin = [-63.5*Trans.spacing,0,startDepth];
```

The first attribute, PData.PDelta specifies the spacing between pixels (in wavelengths) in the X, Y and Z dimensions. The pixel spacing should be chosen to adequately sample the lateral and axial resolution of the ultrasound image, but no finer than necessary. This is

because the higher the pixel density, the longer the image reconstruction will take, since the reconstruction rate in pixels per second is essentially constant. Ultrasound images made with broad bandwidth transducers typically have a best axial resolution on the order of a wavelength and a best lateral resolution of several wavelengths. This means that two points that are spaced smaller than these distances in the axial or lateral dimensions will have echoes that merge together, so that the points appear as a single reflector. To sample the best axial and lateral resolution of the majority of ultrasound transducers with our pixel array, it is usually sufficient to use a pixel spacing in the lateral direction of one wavelength (which is typically close to the element spacing), and in the axial direction of half a wavelength. These are the values set for `PData.PDelta(1)` and `PData.PDelta(3)`.

[Note: The pixel density for the `PData` pixel array is not the same as the pixel density used for displaying the ultrasound image. Most displays have a very high resolution of pixels and thus require scaling up the pixel density for display. This scaling up is similar in some ways to the interpolation performed by scan conversion, although scaling from one rectangular format to another requires very little processing time and is usually performed automatically by the Graphics Processing Unit (GPU) of the computer.]

The next attribute, `PData.Size`, sets the dimensions of the `PData` array in terms of number of rows and number of columns. These number of rows and columns will determine the overall size of the pixel data array in the image field. It is useful to specify the row size in terms of the `startDepth` and `endDepth` of the scan, and in the example above, we have set the number of rows to match the height of the scan region. The number of columns is set to match the number of elements in the width of the transducer. The third dimension of the `PData` array is reserved for three dimensional pixel arrays (arrays of voxels), and here is set to 1.

The last attribute in our example is the `PData.Origin` attribute, which sets the location of the upper left hand corner of our pixel data array in the transducer coordinate system. In this case, we have set the `Origin` to match the center of the first element of our linear array transducer. The `PData` array as defined places a pixel column directly underneath the center of each element in our array. This typical `PData` region is shown in figure 12.2 below.

In general, the `PData` array defines a region large enough to include the area of the medium that can be scanned with a particular transducer. However, this concept is not enforced and the `PData` region can be positioned anywhere in the transducer coordinate system. If the `PData` region is positioned outside the field of view of the transducer, little or no acoustic energy will reach the pixels being used for image reconstruction and the resulting reconstructed image will probably be unusable. It is up to the user to make sure that the `PData` region specified is an adequate region for image reconstruction, both in terms of transmit insonification and receive element sensitivity.

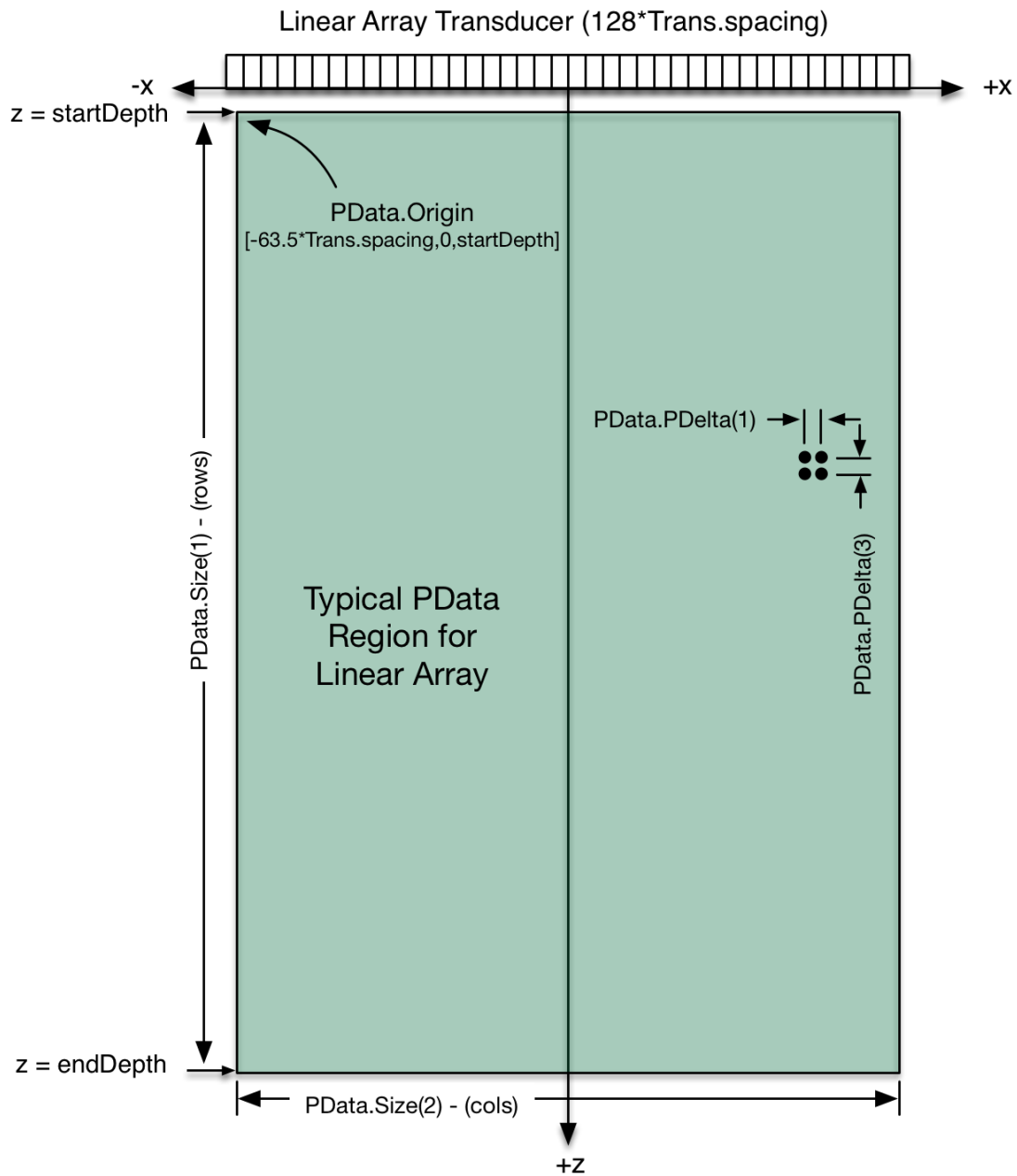


Fig. 12.2 A typical PData region for a linear array transducer. The pixels are aligned with the field of view of the transducer.

While it is possible to specify a PData array that selects a specific local region of the field of view for processing, it is preferable to use the PData.Region definition for this purpose. If the processing region is to be moved around or sized, or there are multiple local regions to process for an image frame, the Region definition will be easier to manage. PData.Regions can be defined with geometric shapes, or arbitrarily in terms of a list of linear indices of the PData array. A number of geometric shapes can be defined, using a 'Shape' structure. For example, to define a rectangular PData.Region, the following definition might be used:

```
PData.Region = struct('Shape', struct('Name', 'Rectangle', 'Position', [0, 0, 20], ...  
                                     'Width', 64*Trans.spacing, 'height', 100));
```

When processed by the utility function, 'computeRegions', the resulting PData.Region will be specified as shown below.

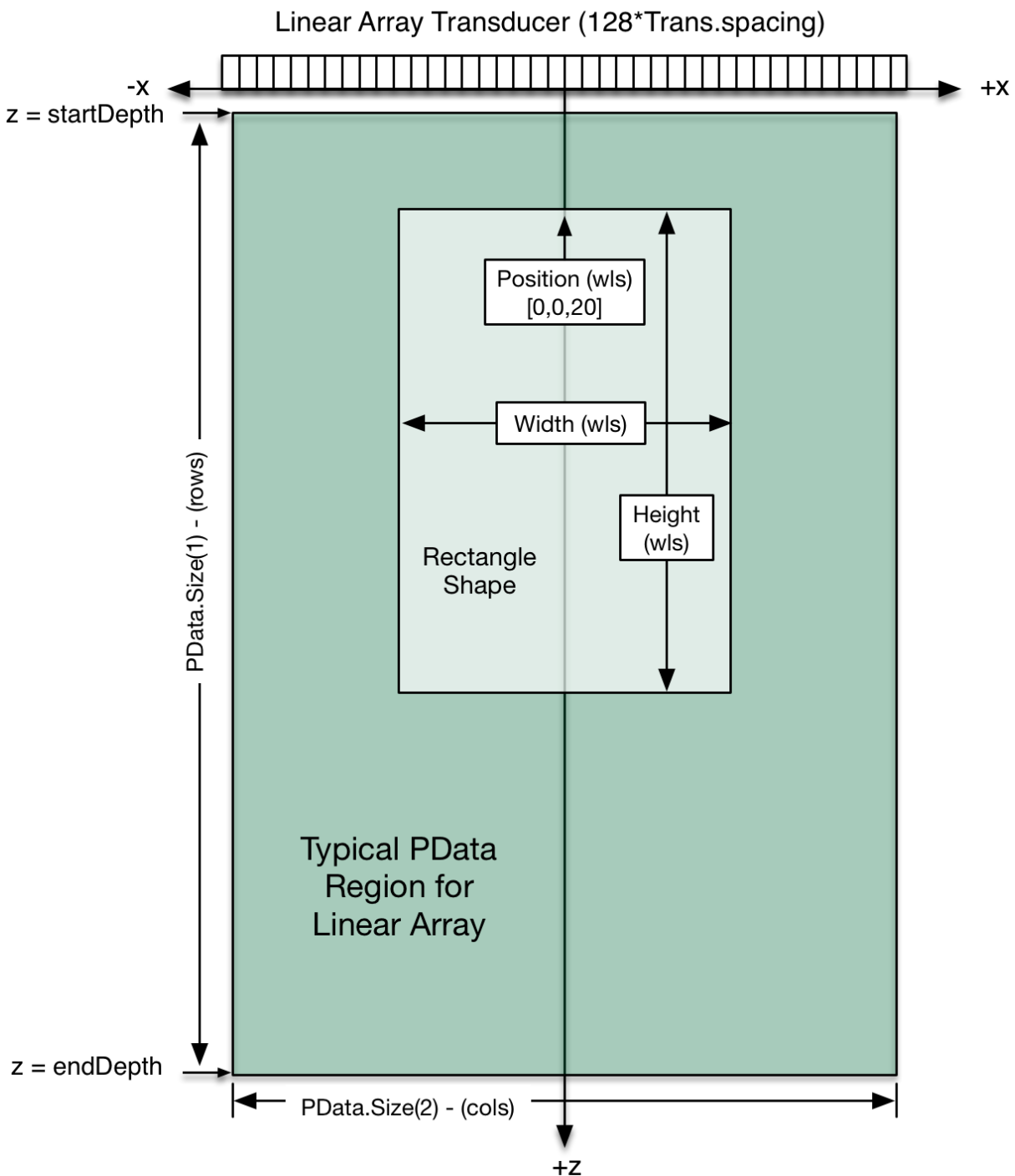


Fig. 12.3 A Rectangle Shape Region within the PData area.

Why would we want a PData.Region that covers only a portion of the field of view of the transducer? We might want to match a processing region with the area insonified by the transmit beam, or we might want to only process data from a region of the image to reduce processing requirements, as in the case of color Doppler imaging. Or, we might want to simply focus in on specific detail in the image, with perhaps a higher pixel density. Note: If no PData.Region is specified, the computeRegions utility function will create a default Region, which is the same size as the PData array.

The computeRegion utility function is called automatically by the loader program, VSX, if not called in the SetUp script. However, calling computeRegions in the SetUp script is often useful if one wants to verify the Regions created before running the program. In our example, with no specified shape Region, we can explicitly call computeRegions with the below statement following the PData definition.

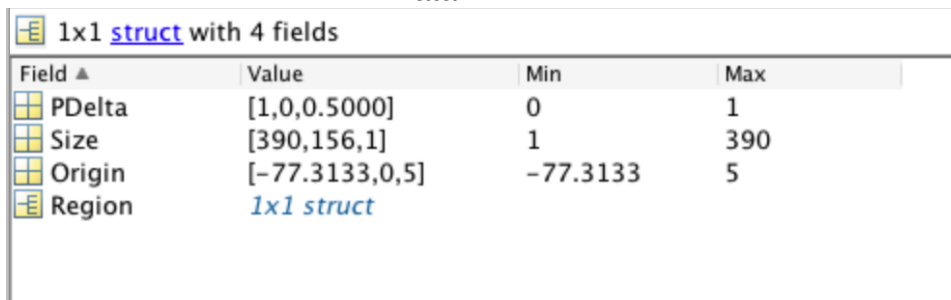
```
% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L7-4 transducer is 'known' transducer.

% Specify PData structure array.
PData.PDelta = [1.0,0,0.5] ;
startDepth = 5;
endDepth = 200;
PData.Size(1) = ceil((endDepth-startDepth)/PData.PDelta(3));
PData.Size(2) = ceil(128*Trans.spacing/PData.PDelta(1));
PData.Size(3) = 1; % 2D image plane
% PData.Origin is the location [x,y,z] of the upper lft corner of the array.
PData.Origin = [-63.5*Trans.spacing,0,startDepth];

PData.Region = computeRegions(PData);
```

Examining the PData variable in the Matlab workspace with the Variable Editor, we see the following attributes:

PData



Field ▲	Value	Min	Max
PDelta	[1,0,0.5000]	0	1
Size	[390,156,1]	1	390
Origin	[-77.3133,0,5]	-77.3133	5
Region	1x1 struct		

The computeRegions utility function has added the PData.Region attribute, which when expanded in the Matlab Variable Editor, shows the attributes listed below.

PData.Region

1x1 struct with 3 fields

Field ▲	Value	Min	Max
Shape	1x1 struct		
numPixels	60840	60840	60840
PixelsLA	1x60840 int32	0	60839

If we examine the Shape structure, we will see that it contains a single attribute, 'Name', which has the value 'PData'. This is the default shape when no PData.Region is specified. The next two attributes give the number of pixels in the Region, and the linear addresses of the PData array that are included in the shape. In this case, all the PData pixels are included. Note that the linear addresses start from 0 rather than 1, and that the datatype is 'int32'. If the user wishes to design their own custom shape, the Name field of the Shape structure is set to 'Custom' and the user then is responsible for setting the numPixels and PixelsLA attributes. In this case, there would be no need to call the computeRegions function. However, in almost all scan formats, the provided Shape definitions should prove adequate. [Note: The PData.Region is no longer divided up into separate sections for processing purposes, as this is now done automatically by the system software.]

If you would like to see visually how the computeRegions.m function computed the PData region(s), you can execute the function, 'showRegions.m' after the regions have been computed, with an argument of PData(n), where n identifies the index of the PData object you would like to use. For example, with only one PData object, you could type:

```
>> showRegions(PData)
```

This function will open a figure window and proceed to graphically show each Region of the the PData object in succession, each with a different color.

12.4 The Recon and ReconInfo object definitions

With the pixels that we want to process defined by the PData object, we now need to specify the attributes of the reconstruction process, using the Recon and ReconInfo structures. The Recon structure will define the general characteristics of the reconstruction for a PData object, including the source and destination buffers, and which ReconInfo structures to apply to the reconstruction process. The ReconInfo structures specify individual reconstruction actions that apply to the individual regions of the PData object. Since we will need a Image buffer for the output of the reconstruction, we must first define the buffer with a Resource specification.

```
Resource.ImageBuffer(1).numFrames = 10;
```

The statement above specifies an ImageBuffer that has a frame size equal to the PData region, and contains 10 frames. The actual data is contained in a cell array named ImgData, which will be placed in the Matlab workspace when our program exits. [Note: Previous releases required specifying rowsPerFrame and colsPerFrame attributes in the Resource.ImageBuffer specification, but now these attributes, if missing, are matched to PData.Size(1) and PData.Size(2).]

Let's look at the Resource and Recon structure definitions for reconstructing an image from our single plane wave transmit/receive acquisition.

```
% Specify Recon structure array for a 128 channel Vantage system.
Recon = struct('senscutoff', 0.6, ...
              'pdatanum', 1, ...
              'rcvBufFrame', -1, ... % use most recently transferred frame
              'ImgBufDest', [1,-1], ... % auto-increment ImageBuffer each recon
              'RINums', 1);
```

In the above Matlab code, we have used the 'struct' function to specify the attributes of our single Recon structure. The first attribute, senscutoff, requires a little explaining. In our Trans specification, there is a definition of the element sensitivity function, sometimes called the directivity function. It describes the relative sensitivity of an element of the transducer to an echo returning at an angle to the element's normal. The sensitivity in the direction of the normal is defined as 1.0, and the sensitivity falls off as the angle to the normal increases. The senscutoff value is the lowest value at which the reconstruction processing will accept an element's contribution. In other words, for a given pixel in the image field of the transducer, if the angle from an element's normal to the pixel location is such that the value of the Trans.ElementSens function at that angle falls below the senscutoff value, the element's RF signal will not contribute to the image reconstruction. This effectively eliminates an element from participating in a pixel's reconstruction if the element is not sensitive to signals coming from the pixel's location. This is very important in the near field of the transducer, since pixels near the transducer surface can only be 'seen' by a few elements. Adding in the signals from the other elements would only contribute clutter to the reconstruction point. Typical senscutoff values for achieving the best tradeoff between lateral resolution and contrast resolution are in the range of 0.5 to 0.7.

The next attribute, pdatanum, is the number of the PData structure to use for the reconstruction. In this case, we have a single PData structure, so we set the value to 1.

The `rcvBufFrame` attribute is an optional attribute that can be used to override the receive buffer frame specified in the `ReconInfo` structures. The `ReconInfo` structures reference a Receive object which specifies a `RcvBuffer` frame. This frame is the usual source data for the reconstruction process defined by the `ReconInfo` structure. In an asynchronous script (processing is asynchronous with data acquisition), there may be many `RcvBuffer` frames, and the differences in the Receive objects for acquiring RF data for each frame may be only the frame number. If this is the case, we can point the reconstruction processing to the most recent frame acquired by setting the `rcvBufFrame` attribute to -1. This will cause the determination of the most recent frame transferred to the `RcvBuffer` at the start of each reconstruction process, overriding the `RcvBuffer` frame referenced by the `ReconInfo` structures associated with the reconstruction. [Note: If the determination of the most recent frame transferred yields the same frame number as the previous reconstruction with the same Recon, the reconstruction process will wait up to one second for a new frame to appear. After this timeout period, the previous frame will be re-processed, and a warning message issued.]

The next attribute, `ImgBufDest`, sets the destination for the output of the reconstruction. If the mode of the reconstruction is set to output IQ data, we would want the destination to be an `InterBuffer` frame, and we would need an `IntBufDest` attribute. If the mode is set to output intensity (amplitude) data, we want to set a destination frame in the `ImageBuffer`. In some cases where there are multiple reconstruction actions, as in the situation where we are using synthetic aperture acquisitions to acquire the signals for our frame, we might need to specify both an `InterBuffer` and an `ImageBuffer` frame. Each of these two attributes specify a two element array, whose values are the buffer number and the frame number. In the case of the `ImageBuffer` frame destination, instead of a specific frame we can use another negative value of -1 to indicate that the next available frame in the buffer should be used for output. This allows defining a multi-frame `ImageBuffer` that can be used for storing multiple output frames.

The last attribute in our `Recon` structure is the `RINums` array. This array specifies the `ReconInfo` structures associated with this Recon. The `ReconInfo` structures specify a reconstruction action to take on specified regions of the `PData` structure. For our example script, reconstruction only requires a single reconstruction action, and thus only references a single `ReconInfo` structure.

Let's list the `ReconInfo` structure(s) needed to make an image from our plane wave transmit acquisition for a 128 channel Vantage system.

```
% Define ReconInfo structure for a 128 channel system.
ReconInfo = struct('mode', 'replaceIntensity', ... % replace intensity data.
                  'txnum', 1, ...
                  'rcvnum', 1, ...
                  'regionnum', 1);
```

The first attribute is the mode of the reconstruction. There are a number of different modes, providing a number of options for reconstruction and storage of the output. These are specified in Table 12.4 below.

ReconInfo Mode	Processing	Output	
		InterBuffer	ImageBuffer
'replaceIntensity' 0	Reconstruct IQ and intensity data.	Replace IQ data. (if destination specified).	Replace intensity data in buffer.
'addIntensity' 1	Reconstruct intensity data.	—	Add to intensity data in buffer.
'multiplyIntensity' 2	Reconstruct intensity data.	—	Multiply times intensity data in buffer.
'replaceIQ' 3	Reconstruct IQ data.	Replace IQ data in buffer.	—
'accumIQ' 4	Reconstruct IQ data.	Add to IQ data in buffer.	—
'accumIQ_ replaceIntensity' 5	Reconstruct IQ and compute intensity of accumulated IQ data.	Add to IQ data in buffer.	Replace intensity data in buffer.
'accumIQ_ addIntensity' 6	Reconstruct IQ and compute intensity of accumulated IQ data.	Add to IQ data in buffer.	Add to intensity data in buffer.
'accumIQ_ multiplyIntensity' 7	Reconstruct IQ and compute intensity of accumulated IQ data.	Add to IQ data in buffer.	Multiply times intensity data in buffer.

Table 12.4 The various ReconInfo.mode operations.

The next two attributes of the ReconInfo structure, `txnum` and `rcvnum`, specify the TX and Receive structures associated with the reconstruction action. The `Receive(rcvnum)` structure will specify the source frame of the RcvBuffer to be used, unless as indicated earlier, we have defined the `Recon.rcvBufFrame` attribute; In this case, with a `rcvBufFrame` setting of -1, the RcvBuffer frame used for the reconstruction will be the most recent frame transferred to the RcvBuffer.

The last attribute in our ReconInfo structure is the `regionnum` value. This value is the index number of a `PData.Region` structure defined in the PData object. Each ReconInfo structure should reference a single Region number for processing. In this case, the `computeRegions` utility function created a single `PData.Region` structure, so our `regionnum` value should be set to 1.

When a Recon structure is included in a SetUp script, an additional GUI control is created in the center column of the GUI window, called 'Speed Of Sound'. This control modifies the speed of sound value used by the reconstruction software from the value defined in the `Resource.Parameter.speedOfSound` attribute (or the default value of 1540 m/s). The slider value provides the amount of modification of the wavelength: 1.0 for no modification, less than one for a shorter wavelength (and slower speed of sound), and more than one for a longer wavelength (and faster speed of sound). The 'Speed of Sound' control only modifies the wavelength value of the reconstruction processing and doesn't change other wavelength values, such as those specified for the transducer and display. Consequently, it is only intended for fine tuning of the image reconstruction.

13.0 Processing the Reconstructed Data.

After reconstruction processing, our echo intensity output data will be found in an ImageBuffer frame, at the same number of points (pixels) as the PData array. If the ImageBuffer was defined larger than the PData array in either rows and columns, the reconstructed data will be placed at the same linear index values as the PData array. For example, if the ImageBuffer frame has 1024 rows and 512 columns, and the PData array has 512 rows and 256 columns, the reconstructed data will be placed in the first 128 columns of the ImageBuffer. Since we spaced our pixels of the PData array sufficiently close together to capture the best resolution of our ultrasound transducer, the reconstructed image is suitable for display; however, the relatively small pixel count and lack of post processing will not make for a very large or pleasing image for viewing. To put a high quality image up on the display screen of the computer, we typically must perform additional image processing operations, such as scaling, compression, reject, and persistence processing.

Processing operations in the Verasonics are defined by a Process object, which defines a classname for the type of processing, a method to use, and list of parameters that help define additional elements of the processing. For processing image data, we use the classname 'Image', and the method 'imageDisplay'. The main attributes of this Process structure are shown below.

```
Process(1).classname = 'Image';
Process(1).method = 'imageDisplay';
Process(1).Parameters = {'imgbufnum',1,... % number of buffer to process.
                        'framenum',-1,... % (-1 => lastFrame)
                        'pdatanum',1,... % number of PData structure
                        'pgain',1.0,... % image processing gain
                        'reject',3,... % see text
                        'persistMethod','simple',... % 'simple' or 'dynamic'
                        'persistLevel', pers,...
                        'grainRemoval','none',... % 'low','medium','high'
                        'processMethod','none',... % see text
                        'averageMethod','none',... % see text
                        'compressMethod','power',... % 'power' or 'log'
                        'compressFactor', 40,...
                        'mappingMethod','full',... % see text
                        'display',1,... % display image after processing
                        'displayWindow',1}; % number of displayWindow to use
```

The 'imageDisplay' processing is configured by providing a list of parameters. The first two parameters, imgbufnum and framenum defined the frame in the ImageBuffer to be use as input. Here again a value of -1 for the frame number indicates that the most recent frame reconstructed should be processed. The next parameter, pdatanum, is the index of the PData structure that was used for reconstruction. This structure is needed to get the image size (rows and columns) as well as the pixel delta values.

The pgain parameter (short for processing gain) specifies a gain factor to apply to the reconstruction intensity values as they are processed. The reconstruction intensity values are normalized by the number of channels contributing to the pixel reconstruction, but if some or all of the channels have low amplitude signals, the reconstruction values will need to be amplified to produce an adequate display intensity value.

The `reject` attribute determines the amplitude level below which the reconstruction output will be mapped to black on the display. The intensity values above this level and below a fixed maximum intensity limit will be scaled from zero to maximum intensity for further processing. The `reject` attribute is used to remove low level noise from the image.

The next two parameters, `persistMethod` and `persistLevel`, set the method and amount of frame-to-frame persistence. This processing is a weighted average of the frames in the `ImageBuffer` that moves with time. The 'simple' persistence method adds a fraction of the previous weighted average frame's intensity values to (1 minus the same fraction of the new frame) to get a new average.

$$\text{NewAverage} = \text{PL} * \text{PreviousAverage} + (1 - \text{PL}) * \text{NewFrame}$$

where: $\text{PL} = \text{persistLevel}/100$

This leads to a running weighted average of the current and previous frames in the buffer. In our `Process` structure, we predefined the `persistLevel` with the variable `pers`. This is so we can access the setting later in a GUI control used to change the value.

The other option for `persistMethod` is 'dynamic', which is used when there is significant motion in the media being imaged. In this case, the factor `PL` is a vector of 4 values which are selected base on the absolute value of the difference between the intensity values of the old average and the new frame. The maximum difference is divided into 4 ranges, and depending on which range the difference falls in, the corresponding index of `PL` is used. The `PL` values are usually set so that the greater the change in intensity from frame to frame, the lower the persistence level, allowing dynamic motion to be visualized, while static structures are more heavily averaged.

The next attribute, `grainRemoval`, can be used to enable a 3x3 matrix filter that aims to eliminate single pixels that differ significantly from their neighbors. This filter has three settings - 'low', 'medium' and 'high' which offer varying amounts of filtering. An additional 3x3 matrix filter, `processMethod`, can also be invoked, which aims at reducing variation in line structures detected within the filter kernel.

The `averageMethod` attribute allows implementing a running average of the current and previous frames. The available choices are 'none', 'runAverage2', and 'runAverage3', of which the latter two select the running average of the current frame with the previous one or two frames. This allows combining frames with different processing to achieve special effects, such as spatial compounding.

We now come to the `compressMethod` parameter, which requires a little explanation. Compression of the echo intensities in an ultrasound image is generally required for simultaneous visualization of low level scattering return and large specular echoes. The dynamic range of intensities can vary over 60dB (1000 to 1), with the majority of intensities clustered at the low end of the range. Most ultrasound systems use a modified logarithmic compression function; modified, because intensities of 0 need to be mapped to output values of 0, rather than -infinity. The `compressMethod` of `log` provides this modified logarithmic compression with a `compressLevel` that determines the dynamic range of the source data that will be compressed. The `compressMethod` of `power` uses a square root compression function in the processing of intensity values.

The `mappingMethod` parameter specifies the portion of the colormap to be used by the intensity values. The choices are `lowerHalf`, `upperHalf`, and `full`. The `lowerHalf` and `upperHalf` choices are used when the color palette needs to be split between two ultrasound modes, such as 2D and color Doppler. For normal 2D processing, we use the `full` setting.

Before describing the `display` and `displayWindow` processing parameters, we should explain a bit more about the steps in the `imageDisplay` processing chain. When we perform additional image processing on data in an `ImageBuffer`, it is usually desirable not to destroy the original data. For this purpose the software creates a shadow buffer to the `ImageBuffer`, called the `ImageP` buffer. The image processing operations for grain removal and persistence use the `ImageBuffer` as input and the `ImageP` buffer as output. For example, with image persistence set, frames are averaged into the `ImageP` buffer, so that each frame is a weighted average of previous frames. The frames in the `ImageBuffer` are therefore preserved. The `ImageP` buffer has the exact same size as the `ImageBuffer`, and includes the same number of frames. Since the `Image` buffer and `ImageP` buffer pixel density is typically lower than needed for presenting on a high resolution display, we need to perform some pixel interpolation to get to a higher pixel density for display. For this purpose, an additional multiframe buffer, named `DisplayDataTemp`, is also created. This buffer, which is not user accessible, holds the interpolated pixel data, that will be used for the `processMethod` and `averageMethod` functions. Finally, the `DisplayDataTemp` data is processed with the `reject`, `compressMethod` and `mappingMethod` functions into an eight bit `DisplayData` buffer before being sent to the Matlab figure window on the computer display. These processing steps and buffer transfers are shown in figure 13-1 below.

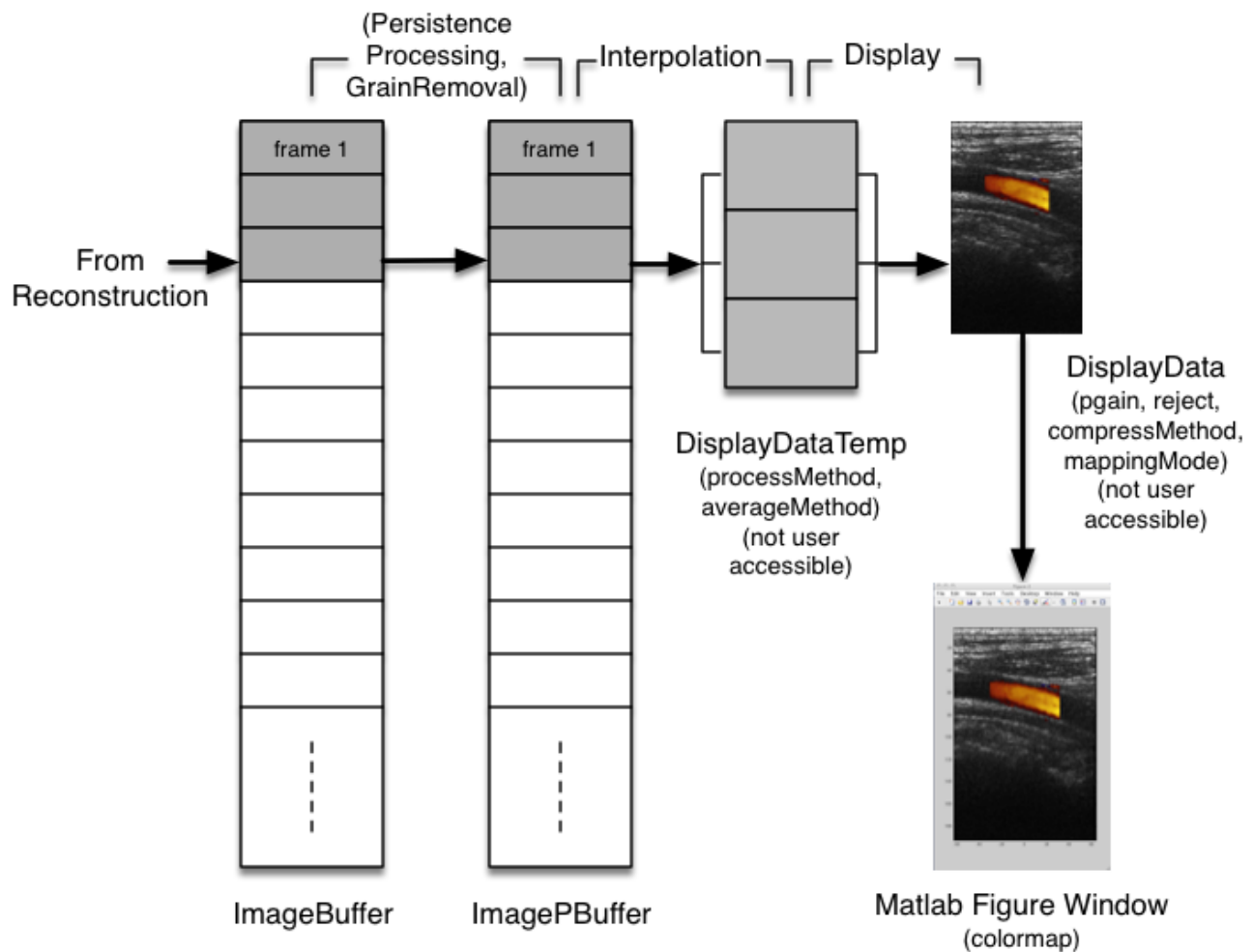


Fig. 13-1 Image processing using the 'imageDisplay' method of the Image class.

The `display` parameter of the Process structure determines whether our processed image data will be written to the Matlab figure window on the computer screen. Sometimes, we may want to just process an image for display without presenting it on the display window. An example is in the 2D and color Doppler scripts where the 2D image is processed first, and then the color Doppler information second. We don't want to display the 2D image until the color Doppler pixels have been processed, so in that case, we can set the `display` parameter to 0 for processing the 2D data. With the `display` parameter set to 0, the image data are processed normally to the DisplayData buffer, but are not copied to the CData of the Matlab figure window.

The `displayWindow` Process parameter is simply the index of a displayWindow structure previously defined in the `Resource.displayWindow` attribute. The displayWindow is needed to define the characteristics of both the DisplayData buffer and the CData buffer of the Matlab figure window. The DisplayWindow defines location and size of the figure window used for display, the pixel resolution, and the colormap used to translate the intensity values into grey levels or colors. In addition, the DisplayData buffers are created at

the same size of the CData buffer for each DisplayWindow. We can choose to have multiple displayWindows, each with their own size, location, pixel delta and colormap.

The format for the DisplayWindow is:

```
Resource.DisplayWindow(1).Title = 'Image Display';
Resource.DisplayWindow(1).pdelta = 0.3;
Resource.DisplayWindow(1).Position = [250,150, ... % lower left corner pos.
    ceil(PData.Size(2)*PData.PDelta(1)/Resource.DisplayWindow(1).pdelta), ...
    ceil(PData.Size(1)*PData.PDelta(3)/Resource.DisplayWindow(1).pdelta)];%height
Resource.DisplayWindow(1).ReferencePt = [PData.Origin(1),0,PData.Origin(3)];
Resource.DisplayWindow(1).AxesUnits = 'mm';
Resource.DisplayWindow(1).Colormap = gray(256);
Resource.DisplayWindow(1).numFrames = 20;
```

The `Resource.DisplayWindow(1).pdelta` attribute is a value that sets the pixel spacing of the display window (in wavelengths). The smaller this value, the larger the image will appear on the computer screen, since more pixels will be created. Note that the pixels on the computer screen are assumed to be equally spaced in both the lateral and vertical dimensions.

The `Resource.DisplayWindow(1).Position` attribute is a 4 element array that defines the lower left coordinate of the display window (in pixels) as well as the width and height. In this case we set the width and height to be in the same proportions as our PData array, but scaled up to the pixel resolution specified by `pdelta`.

The `Resource.DisplayWindow(1).ReferencePt` is a three element array that specifies the location of the display window with respect to the transducer x,y,z coordinate system. In this case it is set to match the location of the PData array.

The default units for the display window axes are wavelengths, but the units can be changed to 'mm' using the `AxesUnits` attribute.

The `Resource.DisplayWindow(1).Colormap` attribute defines a standard colormap array used by Matlab. Here we have chosen the standard 'gray(256)' map, which creates a 256 row, 3 column array (red, green, blue) with equal values that range linearly from 0 to 1.

Finally, the `numFrames` attribute specifies a buffer size for the history of displayed frames. This can be used to capture a cineloop of frames that can be saved to a movie file.

Our Process function is invoked by specifying the Process structure index in the `Event.process` attribute of an Event. With a `framenum` attribute of -1, the last reconstructed frame written to the ImageBuffer is processed each time the Event with the Process specification is executed. When multiple image frames are defined in the DisplayWindow, a slider is placed on the GUI control window that allows viewing previous processed frames in freeze mode.

When an ImageBuffer is defined in the SetUp script, additional default GUI controls appear in the right column of the GUI window. These controls apply to the first Process structure defined that uses the 'Image' class and 'imageDisplay' method. The 'Zoom' and 'Pan' controls work by modifying the PData structure to show different regions of the transducer field. Additional controls for modifying Process structure parameters during run time can be access by opening the PTool control panel from the Tools drop down menu. The 'Cineloop' control appears when there is more than one frame defined for the DisplayWindow and allows stepping backwards through processed frames after a 'freeze'. The 'Save' button saves the current cineloop to an avi file in the current directory.

We can now list the complete example RF acquisition script with the reconstruction and signal processing actions added. The script listed below shows the code for both a Vantage 64 or 128 channel system. Just set the V64 variable to 1 at the top of the script for a 64 channels system; otherwise, leave it set to zero.

14.0 The Complete Acquisition and Image Processing Script

```
% File name SetUpL11_4vAcquireRFandImage.m
% - Asynchronous plane wave transmit acquisition into multiple
%   RcvBuffer frames with reconstruction and image processing.

clear all

V64 = 0; % set to 1 for Vantage 64 system.

startDepth = 5;
endDepth = 200;

% Specify system parameters
Resource.Parameters.numTransmit = 128; % no. of transmit channels.
Resource.Parameters.numRcvChannels = 128-(64*V64); % no. of receive channels.
Resource.Parameters.speedOfSound = 1540; % speed of sound in m/sec
Resource.Parameters.fakeScanhead = 1; % optional (if no L11-4v)
Resource.Parameters.simulateMode = 0; % runs script with hardware

% Specify media points
pt1; % use predefined collection of media points

% Specify Trans structure array.
Trans.name = 'L11-4v';
Trans.units = 'mm';
Trans.frequency = 6.25; % not needed if using default center frequency
Trans = computeTrans(Trans); % L7-4 transducer is 'known' transducer.

% Specify PData structure array.
PData.PDelta = [Trans.spacing,0,0.5]; % x, y and z pixel deltas
PData.Size(1) = ceil((endDepth-startDepth)/PData.PDelta(3));
PData.Size(2) = ceil(128*Trans.spacing/PData.PDelta(1));
PData.Size(3) = 1; % 2D image plane
% PData.Origin is the location [x,y,z] of the upper left corner of the array.
PData.Origin = [-63.5*Trans.spacing,0,startDepth];
PData.Region = computeRegions(PData);

% Specify Resource buffers.
% Specify Resource buffers.
Resource.RcvBuffer(1).datatype = 'int16';
Resource.RcvBuffer(1).rowsPerFrame = 4096; % this allows for 1/4 maximum range
Resource.RcvBuffer(1).colsPerFrame = 128;
Resource.RcvBuffer(1).numFrames = 10; % minimum size is 1 frame.
Resource.InterBuffer(1).numFrames = 1; % InterBuffer needed for V64=1
Resource.ImageBuffer(1).numFrames = 10;
Resource.DisplayWindow(1).Title = 'L11-4v Plane Wave Transmit';
Resource.DisplayWindow(1).pdelta = 0.3;
Resource.DisplayWindow(1).Position = [250,150, ... % lower left corner pos.
    ceil(PData.Size(2)*PData.PDelta(1)/Resource.DisplayWindow(1).pdelta), ...
    ceil(PData.Size(1)*PData.PDelta(3)/Resource.DisplayWindow(1).pdelta)];
Resource.DisplayWindow(1).ReferencePt = [PData.Origin(1),0,PData.Origin(3)];
Resource.DisplayWindow(1).AxesUnits = 'mm';
Resource.DisplayWindow(1).Colormap = gray(256);
Resource.DisplayWindow(1).numFrames = 20;

% Specify Transmit waveform structure.
TW(1).type = 'parametric';
TW(1).Parameters = [6.25,0.67,2,1]; % A, B, C, D

% Specify TX structure array.
TX(1).waveform = 1; % use 1st TW structure.
TX(1).focus = 0;
TX(1).Apod = ones(1,Trans.numelements);
TX(1).Delay = computeTXDelays(TX(1));

% Specify TGC Waveform structure.
```

```

TGC(1).CntrlPts = [300,450,575,675,750,800,850,900];
TGC(1).rangeMax = 200;
TGC(1).Waveform = computeTGCWaveform(TGC);

% Specify Receive structure array -
Receive = repmat(struct(...
    'Apod', zeros(1,128), ...
    'startDepth', 0, ...
    'endDepth', 200, ...
    'TGC', 1, ...
    'mode', 0, ...
    'bufnum', 1, ...
    'framenum', 1, ...
    'acqNum', 1, ...
    'sampleMode', 'NS200BW', ...
    'LowPassCoef', [], ...
    'InputFilter', []), ...
    1, Resource.RcvBuffer(1).numFrames*(V64+1));

% - Set event specific Receive attributes.
if (V64==0)
    for i = 1:Resource.RcvBuffer(1).numFrames
        Receive(i).Apod = ones(1,128);
        Receive(i).framenum = i;
    end
else
    for i = 1:Resource.RcvBuffer(1).numFrames
        % -- 1st synthetic aperture acquisition for full frame.
        Receive(2*i-1).Apod(1:64) = 1.0;
        Receive(2*i-1).framenum = i;
        Receive(2*i-1).acqNum = 1;
        % -- 2nd synthetic aperture acquisition for full frame.
        Receive(2*i).Apod(65:128) = 1.0;
        Receive(2*i).framenum = i;
        Receive(2*i).acqNum = 2; % two acquisitions per frame
    end
end

% Specify Recon structure array for 2 board system.
Recon = struct('senscutoff', 0.6, ...
    'pdatanum', 1, ...
    'rcvBufFrame', -1, ... % use most recently transferred frame
    'IntBufDest', [1,1], ... % needed if V64 = 1
    'ImgBufDest', [1,-1], ... % auto-increment ImageBuffer each recon
    'RINums', 1);

if V64==0
    % Define ReconInfo structure for a 128 channel system.
    ReconInfo(1) = struct('mode', 'replaceIntensity', ...
        'txnum', 1, ...
        'rcvnum', 1, ...
        'regionnum', 1);
else
    % Define ReconInfo structure for a 64 channel system.
    Recon(1).RINums = [1;2];
    ReconInfo(1) = struct('mode', 'accumIQ', ...
        'txnum', 1, ...
        'rcvnum', 1, ...
        'regionnum', 1);
    ReconInfo(2) = struct('mode', 'accumIQ_replaceIntensity', ...
        'txnum', 1, ...
        'rcvnum', 2, ...
        'regionnum', 1);
end

% Specify processing events.
pers = 30;
Process(1).classname = 'Image';
Process(1).method = 'imageDisplay';
Process(1).Parameters = {'imgbufnum', 1, ... % number of buffer to process.

```

```

        'framenum',-1,...    % (-1 => lastFrame)
        'pdatanum',1,...    % number of PData structure
        'pgain',1.0,...     % image processing gain
        'reject',3,...      % see text
        'persistMethod','simple',... % 'simple' or 'dynamic'
        'persistLevel', pers,...
        'grainRemoval','none',... % 'low','medium','high'
        'processMethod','none',... % see text
        'averageMethod','none',... % see text
        'compressMethod','power',... % 'power' or 'log'
        'compressFactor', 40,...
        'mappingMethod','full',... % see text
        'display',1,...     % display image after processing
        'displayWindow',1}; % number of displayWindow to use

Process(2).classname = 'External';
Process(2).method = 'myProcFunction';
Process(2).Parameters = {'srcbuffer','receive',... % name of buffer to process.
                        'srcbufnum',1,...
                        'srcframenum',-1,... % process the most recent frame.
                        'dstbuffer','none'};

% Specify sequence events.
SeqControl(1).command = 'timeToNextAcq';
SeqControl(1).argument = 10000;
SeqControl(2).command = 'jump';
SeqControl(2).argument = 1;
SeqControl(3).command = 'timeToNextAcq'; % time between syn. aper. acquisitions
SeqControl(3).argument = 500;

nsc = 4; % start index for new SeqControl

n = 1; % start index for Events
for i = 1:Resource.RcvBuffer(1).numFrames
    Event(n).info = 'Acquire RF Data.';
    Event(n).tx = 1; % use 1st TX structure.
    Event(n).rcv = (V64+1)*i-V64; % use 1st Rcv structure for frame.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 0; % no processing

    if V64==1
        Event(n).seqControl = 3;
        n = n+1;
        Event(n).info = 'Acquire RF Data for 2nd half of aperture.';
        Event(n).tx = 1; % use 1st TX structure.
        Event(n).rcv = 2*i; % use Rcv structure for frame.
        Event(n).recon = 0; % no reconstruction.
        Event(n).process = 0; % no processing
    end

    Event(n).seqControl = [1,nsc]; % time between frames and transfer
    SeqControl(nsc).command = 'transferToHost';
    nsc = nsc + 1;
    n = n+1;

    Event(n).info = 'Call external Processing function.';
    Event(n).tx = 0; % no TX structure.
    Event(n).rcv = 0; % no Rcv structure.
    Event(n).recon = 0; % no reconstruction.
    Event(n).process = 2; % call ext. processing function
    Event(n).seqControl = 0;
    n = n+1;

    Event(n).info = 'Perform reconstruction and image display processing.';
    Event(n).tx = 0; % no TX structure.
    Event(n).rcv = 0; % no Rcv structure.

```

```

    Event(n).recon = 1;      % reconstruction.
    Event(n).process = 1;   % call image processing function
    Event(n).seqControl = 0;
    n = n+1;
end
Event(n).info = 'Jump back to Event 1.';
Event(n).tx = 0;           % no TX structure.
Event(n).rcv = 0;          % no Rcv structure.
Event(n).recon = 0;        % no reconstruction.
Event(n).process = 0;      % no processing
Event(n).seqControl = 2; % jump back to Event 1.

% - Create UI controls for channel selection
nr = Resource.Parameters.numRcvChannels;
UI(1).Control = {'UserB1','Style','VsSlider',...
    'Label','Plot Channel',...
    'SliderMinMaxVal',[1,128-64*V64,64-32*V64],...
    'SliderStep',[1/nr,8/nr],...
    'ValueFormat','%3.0f'};
UI(1).Callback = {'assignin(''base'', ''myPlotChnl'',round(UIValue))'};
EF(1).Function = text2cell('%EF#1');

% Save all the structures to a .mat file.
save('L11-4vAcquireRF');
return

%EF#1
myProcFunction(RData)
persistent myHandle
% If 'myPlotChnl' exists, read it for the channel to plot.
if evalin('base','exist(''myPlotChnl'', ''var'')')
    channel = evalin('base','myPlotChnl');
else
    channel = 32; % Channel no. to plot
end
% Create the figure if it doesn't exist.
if isempty(myHandle) || ~ishandle(myHandle)
    figure;
    myHandle = axes('XLim',[0,1500],'YLim',[-4096 4096], ...
        'NextPlot','replacechildren');
end
% Plot the RF data.
plot(myHandle,RData(:,channel));
drawnow
%EF#1

```

This listing can be copied and pasted into a Matlab editor window and saved to the ExampleScripts folder if desired. Be sure to set the V64 parameter to 1 if executing on a Vantage 64 system. When run in simulate mode, the sequence will display two figure windows in addition to the GUI window, one for the 2D image, and the other for plotting the RF signal from the channel selected.

This example script is now similar to the SetUpL11_4vFlashExtFunc.m script in the ExampleScripts folder. The 2D image, generated with a single plane wave transmit, does not provide the best image quality, but demonstrates some of the flexibility and power of the pixel-oriented reconstruction method. Even this simple example script offers users a number of possibilities for experimenting with ultrasound acquisition and processing, through modification of such factors as the transmit waveform and apodization function, the receive bandwidth, the spacing of reconstruction pixels, the reconstruction sensitivity cutoff, the persistence processing, and the compression curve. The user is encouraged to experiment with the many parameters in this script, and to study the methods used by other

example scripts included in the ExampleScripts folder. When your sequence writing skills are fully developed, you will be able to explore the field of ultrasound imaging wherever your interest takes you.