Proteus DUC Primer

Rev. 1.1

Warranty Statement

Products sold by Tabor Electronics Ltd. are warranted to be free from defects in workmanship or materials. Tabor Electronics Ltd. will, at its option, either repair or replace any hardware products which prove to be defective during the warranty period. You are a valued customer. Our mission is to make any necessary repairs in a reliable and timely manner.

Duration of Warranty

The warranty period for this Tabor Electronics Ltd. hardware is one year, except software and firmware products designed for use with Tabor Electronics Ltd. Hardware is warranted not to fail to execute its programming instructions due to defect in materials or workmanship for a period of ninety (90) days from the date of delivery to the initial end user.

Return of Product

Authorization is required from Tabor Electronics before you send us your product for service or calibration. Call your nearest Tabor Electronics support facility. A list is located on the last page of this manual. If you are unsure where to call, contact Tabor Electronics Ltd. Tel Hanan, Israel at 972-4-821-3393 or via fax at 972-4-821-3388. We can be reached at: support@tabor.co.il

Limitation of Warranty

Tabor Electronics Ltd. shall be released from all obligations under this warranty in the event repairs or modifications are made by persons other than authorized Tabor Electronics service personnel or without the written consent of Tabor Electronics.

Tabor Electronics Ltd. expressly disclaims any liability to its customers, dealers and representatives and to users of its product, and to any other person or persons, for special or consequential damages of any kind and from any cause whatsoever arising out of or in any way connected with the manufacture, sale, handling, repair, maintenance, replacement or use of said products. Representations and warranties made by any person including dealers and representatives of Tabor Electronics Ltd., which are inconsistent or in conflict with the terms of this warranty (including but not limited to the limitations of the liability of Tabor Electronics Ltd. as set forth above), shall not be binding upon Tabor Electronics Ltd. unless reduced to writing and approved by an officer of Tabor Electronics Ltd. This document may contain flaws, omissions, or typesetting errors. No warranty is granted nor liability assumed in relation thereto. The information contained herein is periodically updated and changes will be incorporated into subsequent editions. If you have encountered an error, please notify us at support@taborelec.com. All specifications are subject to change without prior notice. Except as stated above, Tabor Electronics Ltd. makes no warranty, express or implied (either in fact or by operation of law), statutory or otherwise; and except to the extent stated above, Tabor Electronics Ltd. shall have no liability under any warranty, express or implied (either in fact or by operation of law), statutory or otherwise.

Proprietary Notice

Document Revision History

Revision	Date	Description	Author
1.1	18-Jul-2025	• Figure 1.2: Figure b) now shows the distance between the Nyquist bands.	Joan Mercade Jakob Apelblat
1.0	28-Jun-2023	• Original version.	Joan Mercade

Acronyms & Abbreviations

Acronym	Description
µs or us	Microseconds
ADC	Analog to Digital Converter
AM	Amplitude Modulation
ASIC	Application-Specific Integrated Circuit
ATE	Automatic Test Equipment
AWG	Arbitrary Waveform Generators
AWT	Arbitrary Waveform Transceiver
BNC	Bayonet Neill–Concelm (coax connector)
BW	Bandwidth
CW	Carrier Wave
DAC	Digital to Analog Converter
dBc	dB/carrier. The power ratio of a signal to a carrier signal, expressed in decibels
dBm	Decibel-Milliwatts. E.g., 0 dBm equals 1.0 mW.
DDC	Digital Down-Converter
DHCP	Dynamic Host Configuration Protocol
DSO	Digital Storage Oscilloscope
DUC	Digital Up-Converter
ENoB	Effective Number of Bits
ESD	Electrostatic Discharge
EVM	Error Vector Magnitude
FPGA	Field-Programmable Gate Arrays
GHz	Gigahertz
GPIB	General Purpose Interface Bus
GS/s	Giga Samples per Second
GUI	Graphical User Interface
HP	Horizontal Pitch (PXIe module horizontal width, 1 HP = 5.08mm)
Hz	Hertz
IF	Intermediate Frequency
I/O	Input / Output
IP	Internet Protocol
IQ	In-phase Quadrature
IVI	Interchangeable Virtual Instrument
JSON	JavaScript Object Notation
kHz	Kilohertz

Acronym	Description
LCD	Liquid Crystal Display
LO	Local Oscillator
MAC	Media Access Control (address)
MDR	Mini D Ribbon (connector)
MHz	Megahertz
MIMO	Multiple-Input Multiple-Output
ms	Milliseconds
NCO	Numerically Controlled Oscillator
ns	Nanoseconds
PC	Personal Computer
PCAP	Projected Capacitive Touch Panel
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PRBS	Pseudorandom Binary Sequence
PRI	Pulse Repetition Interval
PXI	PCI eXtension for Instrumentation
PXIe	PCI Express eXtension for Instrumentation
QC	Quantum Computing
Qubits	Quantum bits
RADAR	Radio Detection And Ranging
R&D	Research & Development
RF	Radio Frequency
RT-DSO	Real-Time Digital Oscilloscope
s	Seconds
SA	Spectrum Analyzer
SCPI	Standard Commands for Programmable Instruments
SFDR	Spurious Free Dynamic Range
SFP	Software Front Panel
SMA	Subminiature version A connector
SMP	Subminiature Push-on connector
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TFT	Thin Film Transistor
T&M	Test and Measurement
TPS	Test Program Sets
UART	Universal Asynchronous Receiver-Transmitter
USB	Universal Serial Bus
VCP	Virtual COM Port
Vdc	Volts, Direct Current
V p-p	Volts, Peak-to-Peak
VSA	Vector Signal Analyzer
VSG	Vector Signal Generator
WDS	Wave Design Studio

Contents

Figures

Tables

1 Introduction

The implementation of the DUC (Digital Up-Converter) in the Proteus family of products is depicted in the figure 1.1 below.

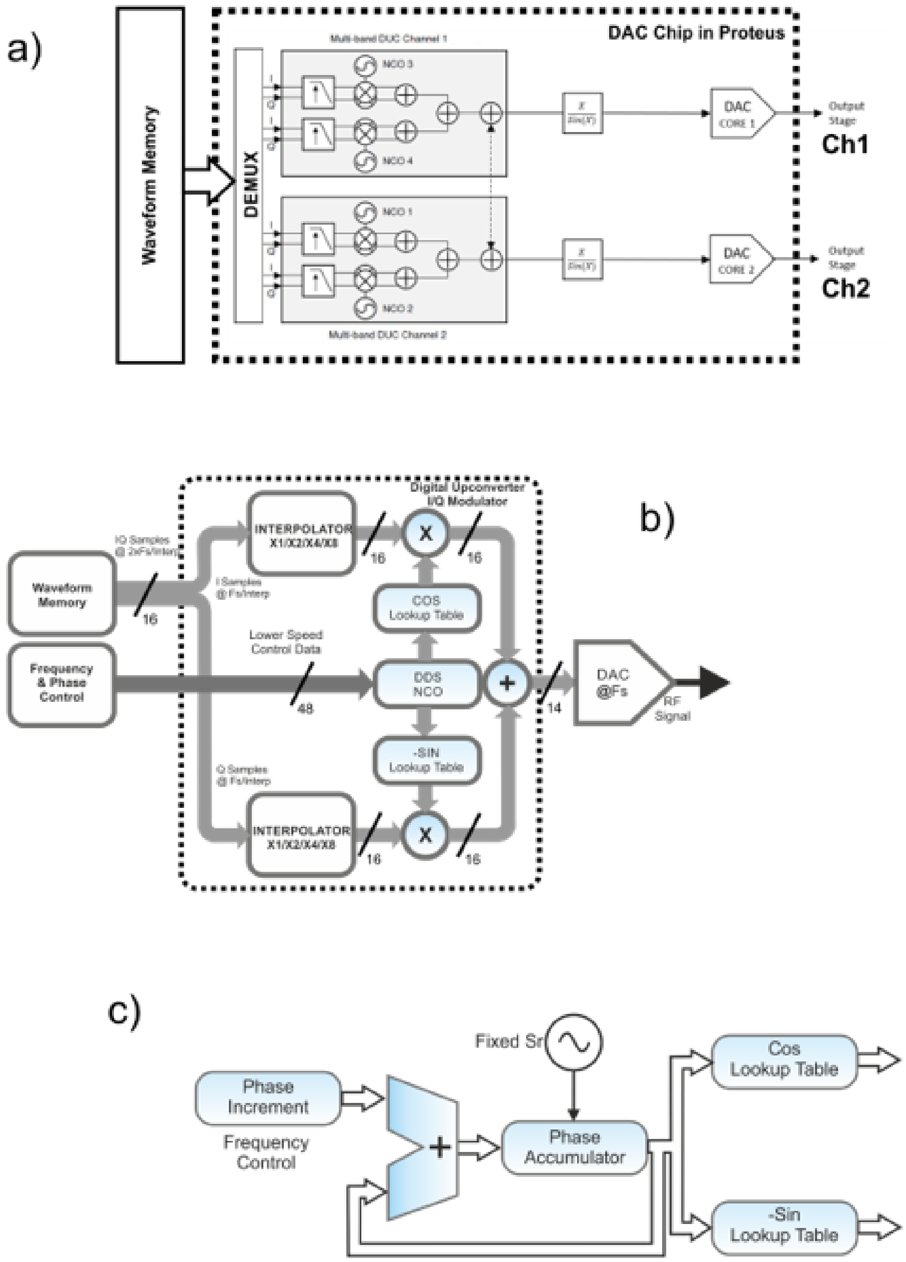


Figure 1.1 Proteus modules incorporate one or two two-channel DAC chips. Each chip is associated to a Waveform Memory bank. In the DUC mode, each channel is associated to two independent DUCs (a). Each DUC is an Numerical IQ modulator with its own Numerically Controlled Oscillator acting as the L.O. for the modulator (b). The NCO is implemented as a DDS synthesizer where frequency and phase can be controlled through two registers (c). In Proteus the DDS Frequency Control register is 48-bit wide so frequency can be set with 30µHz resolution at maximum SR_{DAC} (9 GS/s).

Two independent DUC blocks are associated to each AWG channel. The DUC functionality is standard in the P948X family and optional in the P258X. The theory of the DUC and the advantages of using it for RF signal generation can be read in the “Proteus Programming Manual”. Proteus can use one of the DUCs for a given channel or both simultaneously. The trade-off is the maximum modulation bandwidth (using two DUCs results in half of the available modulation BW). Additionally, an additional IQ modulation mode combines the processing part of two channels to produce one single output, so just half of the processing chain for one of the DUC in each channel is used. In this way, modulation bandwidth doubles compared to using one full DUC for each channel.

Table 1.1 Maximum DAC Sampling Rate and Modulation Bandwidth vs. DUC Mode and Interpolation Factor

DUC Mode	Interpolation X2	Interpolation X4	Interpolation X8
IQ MODE HALF (Half DUC/Channel)	Max SR_{DAC}: 5GS/s Mod. BW = 2.5GHz	Max SR_{DAC}: 9GS/s Mod. BW = 2.25GHz	Max SR_{DAC}: 9GS/s Mod. BW = 1.125GHz
IQ MODE ONE (One DUC/Channel)	Max SR_{DAC}: 2.25GS/s Mod. BW = 1.25GHz	Max SR_{DAC}: 5GS/s Mod. BW = 1.25GHz	Max SR_{DAC}: 9GS/s Mod. BW = 1.125GHz
IQ MODE TWO (Two DUCs/Channel)	Max SR_{DAC}: 1.25GS/s Mod. BW = 625MHz	Max SR_{DAC}: 2.5GS/s Mod. BW = 625MHz	Max SR_{DAC}: 5GS/s Mod. BW = 562.5MHz

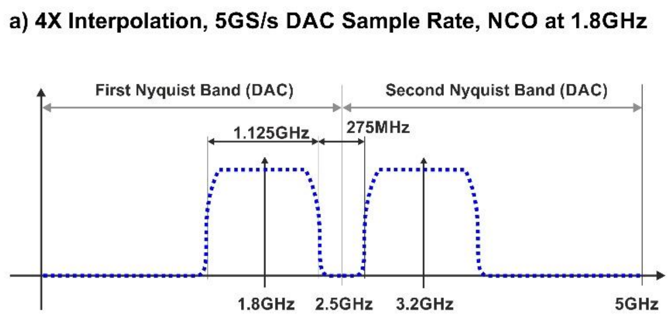
It is important to understand how the baseband I/Q waveforms are transformed in a fully modulated RF signals ready for digital-to-analog conversion. In this way, the right settings can be set in the generator and the right parameters can be applied to the calculation of the baseband waveforms. I and Q waveforms are stored in the waveform memory and supplied to the DUC processing chain at some integer fraction of the final sampling rate. I and Q samples always use 16-bit samples and the overall transfer rate for each channel cannot go beyond 5GBytes/s. As IQ modulation takes place at the final sampling rate, the sampling rate of the incoming IQ data must be interpolated by 1X, 2X, 4X, or 8X interpolation factors. For a given DAC sampling rate, only interpolation factors resulting in data transfer rate per channel equal or lower than the maximum, are acceptable. The above table gives the maximum DAC sampling rate and Modulation Bandwidth depending on the DUC mode and interpolation factor. Sampling rate for the baseband signals (SR_{BB}) can be calculated as a fraction of the DAC sampling rate (SR_{DAC}) and the interpolation factor (IF) using the following expression:

$$SR_{BB} = SR_{DAC} / IF \qquad (1)$$

The data throughput (DT), expressed in bits/s, from the waveform memory for a given IQ pair can be calculated using the formula below:

$$DT = SR_{BB} \times 2 \text{ samples} \times 2 \text{ bytes} = SR_{BB} \times 4 \qquad (2)$$

Let’s use an example for the generation of a 1GHz BW, modulated signal at 1.8GHz carrier frequency using a P9484M in the IQ Mode ONE, so only one of the two DUCs is used, see figure 1.4b below.



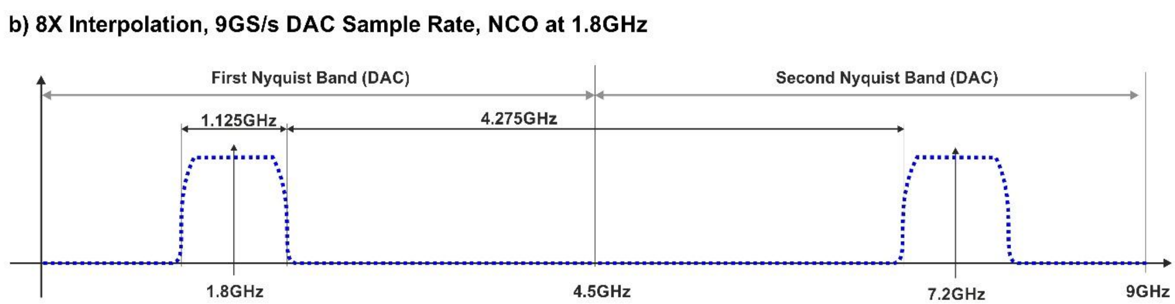


Figure 1.2 2 Real-Time interpolation is applied in Proteus to reduce the sampling rate of the baseband I/Q waveforms while keeping the high SR_{DAC} required to obtain a high enough Nyquist Frequency and sufficient image separation. 2X, 4X, and 8X are the interpolation factors implemented in the Proteus unit. In a), with SR_{DAC} = 5GS/s, IF = 4X, MB = 1.125GHz, and FC = 1.8GHz results in images in the first NZ and the second NZ separated by just 275MHz. In b), Increasing SR_{DAC} to 9GS/s and setting the IF to 8X, the separation between images is 4.275GH.

If we take the DAC sampling rate to be 5GS/s and interpolation factor 2X, the baseband sampling rate will be:

SR_{BB} = 5GHz / 2 = 2.5GS/s

And the data throughput will be:

DT = 2.5GS/s x 4 Bytes / complex sample = 10GBps > 5GBps

This means that the 2X interpolation factor cannot be used in this case, if we do the same calculations with the 4X interpolation factor, we obtain

SR_{BB} = 5GHz / 4 = 1.25GS/s, DT = 1.25GS/s x 4 bytes / complex sample = 5GBps <= 5GBps

which is within the operational limits of Proteus. In this case, modulation BW will be close to 1.25GHz, which is greater than the required 1GHz. The closest image at the output of the DAC will be located at 5GHz-1.8GHz = 3.2GHz, and the gap between the image in the first NZ and the image in the second NZ will be just 400MHz as shown in the figure 1.2b.

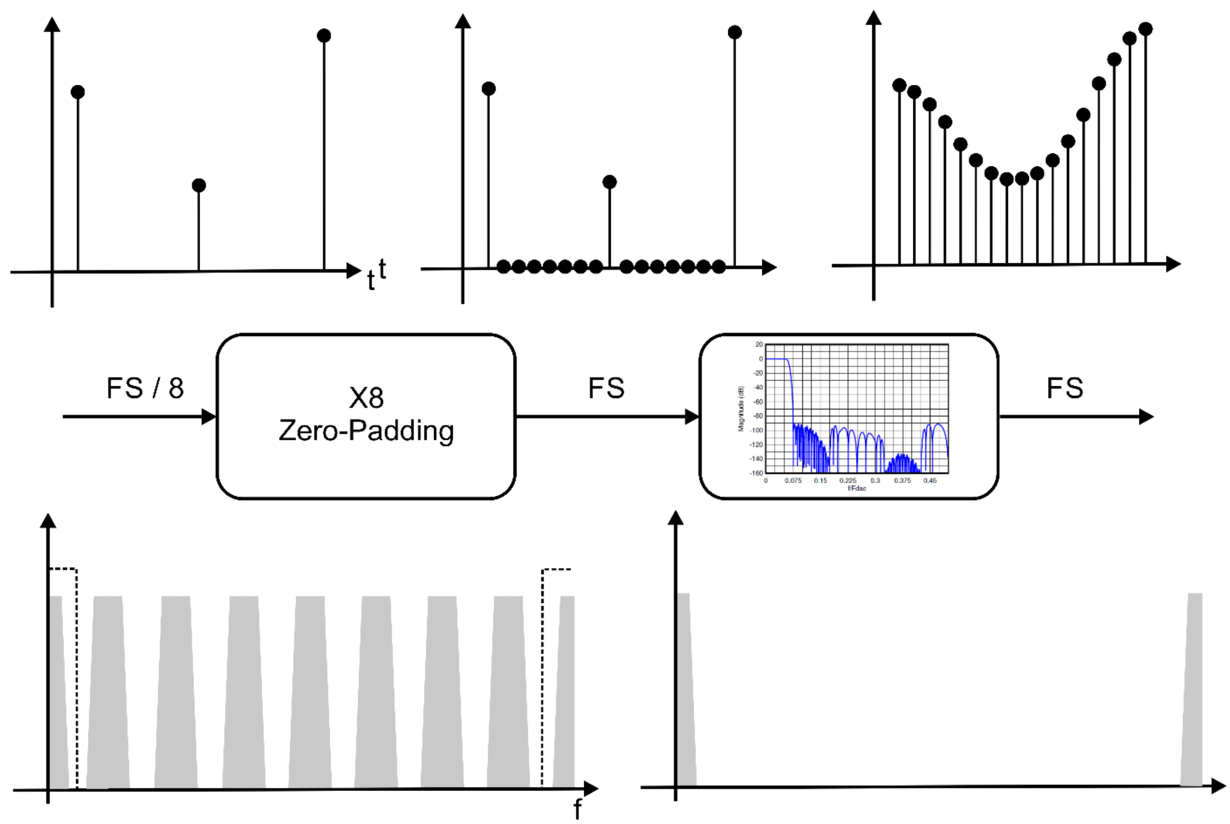


Figure 1.3 Interpolation in real-time is carried out by applying xN zero padding process so the sampling rate is multiplied by factor N. Then, a near-ideal low-pass filter is applied to remove all the unwanted images so only the original first NZ signal (f < SR_{BB} / 2) and the new image close to SR_{DAC} are preserved.

Removing the unwanted image (the one in the second NZ in this case) would require a complex and expensive band-pass filter. In order to avoid this issue, the DAC sampling rate could be increased to the maximum for the P9484M, 9GS/s. In this case, using the 4x interpolator would result in an overall 9GBps data throughput, beyond the operational limits of this unit, so the interpolation factor must be set to the maximum 8x (as shown in fig. 1.3). This results in

SR_{BB} = 9GHz / 8 = 1.125GS/s, DT = 1.125GS/s x 4 bytes / complex sample = 4.5GBps < 5GBps

In this case, modulation BW will be close to 1.125GHz, still larger than the required 1GHz. However, the image in the second NZ will be located at 9GHz – 1.8GHz = 7.2GHz. The gap between the images in the first and the second NZs will be 4.4GHz now, so filtering out the unwanted image will be much simpler. Interpolators are implemented in the way shown in the figure 1.3.

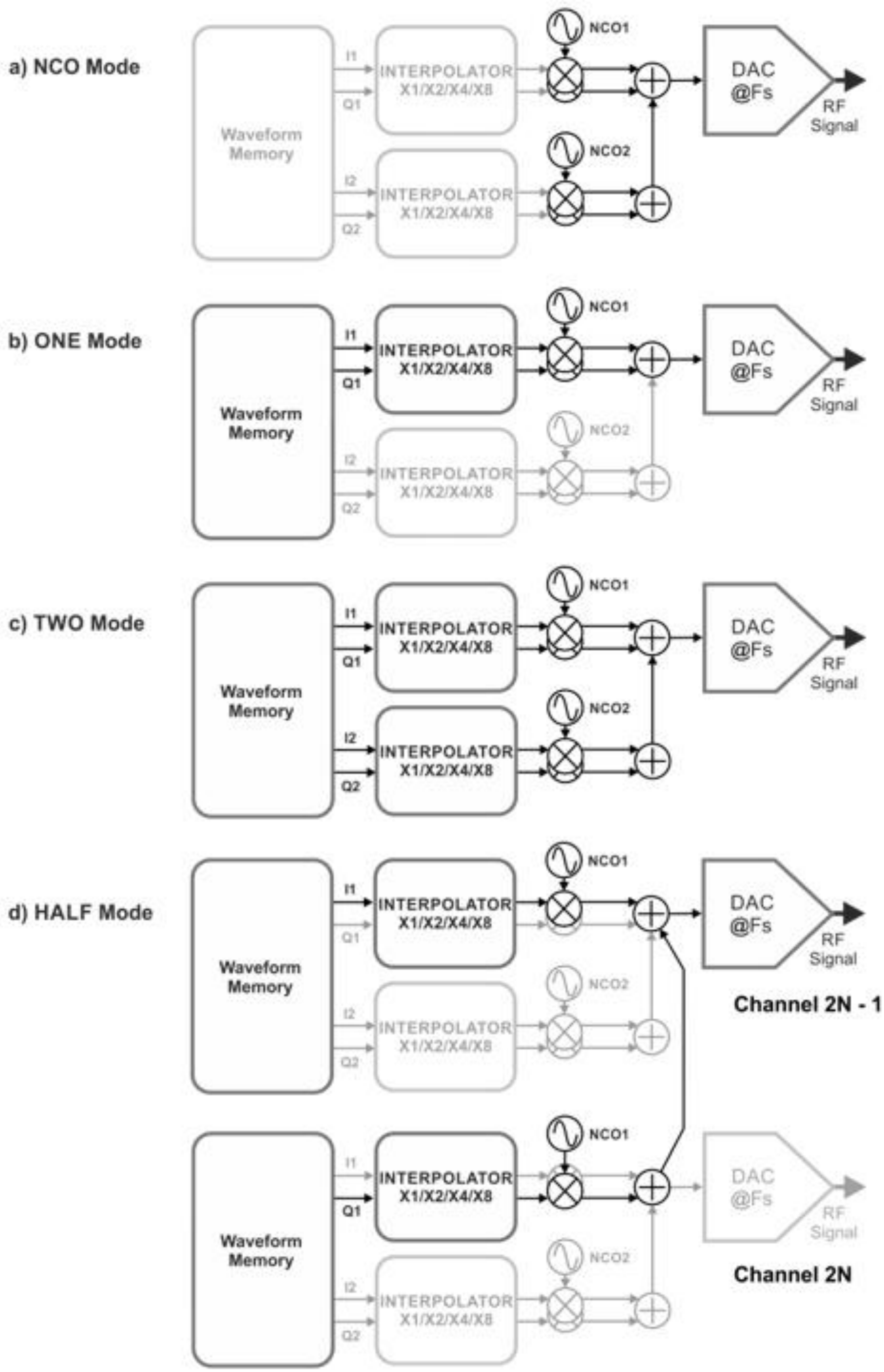


Figure 1.4 In Proteus, The DUC can work in four modes. In the NCO Mode (a), there is no modulation, and the NCOs can be used to generate carriers at any frequency. In the IQ Mode ONE (b), just one of the DUCs is used and just one IQ waveform is read from the waveform memory. The IQ Mode TWO (c) uses both DUCs to produce two independent modulated carriers. It requires two multiplexed IQ waveforms sampled at the same SR_{BB}. Finally, the HALF mode use the DUC infrastructure in two channels so SR_{BB} can be increased by a factor of two, at the expense of disabling half of the channels.

First, sampling rate is increased through a process called zero-padding. It adds N-1 zeros after each sample for a xN interpolation factor. Then, a FIR interpolation digital filter is applied to remove the images of the original non-interpolated waveform and leave just the ones corresponding to the new sampling frequency. As real-world filters are not like the ideal brick-wall low-pass filter, the available modulation BW is slightly lower than the sampling rate before interpolation. Typically, actual modulation BW will be around 80 to 95% of the SR_{BB} expressed in Hz.

The DUC is actually a numerical IQ modulator. As in any IQ modulator, two sinewaves with 90^{O} phase difference have to be supplied to the multipliers applied to the I and Q interpolated waveforms. In Proteus, these sinewaves come from a single DDS (Direct Digital Synthesis) architecture NCO (Numerically Controlled Oscillator) as shown in figure 1.1c. The same phase accumulator value is applied to two different lookup tables where a Cos(x) and a -Sin(x) waveform are implemented. In this way, quadrature balance and error, and carrier feedthrough are perfect as numerical process does not leave room to any of these impairments. In Proteus, the NCO can be set to any frequency between DC and the current sampling rate, so it covers the first two Nyquist Zones. This is important when the wanted image is in one of the even-numbered Nyquist Zones as it allows for the correct generation of modulated signals without inverting the spectrum of the incoming baseband signal (this can be accomplished by inverting one of the components or by swapping the I and the Q components). Both operations would require updating the waveforms in the memory when switching from an even to an odd numbered NZ). As in any DDS synthesizer, the frequency is set by loading a control binary word in a frequency-control register. Phase can also be controlled through a phase-control register. The size of the frequency-control register also defines the frequency resolution of the DDS. In the Proteus DUC, the DDS in the NCO uses a 48-bit frequency-control register. The output frequency of the DDS can be calculated through the following expression:

$$F_{OUT} = SR_{DAC} * k / 2^N,\ k = 0 \ldots 2^N - 1 \qquad (3)$$

where N is the size of the frequency-control register expressed in bits. Frequency resolution will be

$$F_{RES} = SR_{DAC} / 2^N \qquad (4)$$

For the 48-bit frequency-control register in Proteus, F_{RES} @ 9GS/s is 32μHz.

As mentioned earlier, Proteus incorporates two DUCs per channel. They can be used together in the IQ mode TWO (Figure 1.4c). The way to calculate the baseband sampling rates remain the same. However, as two IQ pair waveforms are fed into the same channel, expression (2) above has to be modified to the following:

$$DT = SR_{BB} \text{ x } 2 \text{ IQ pair x } 2 \text{ samples x } 2 \text{ bytes} = SR_{BB} \text{ x } 8 \qquad (5)$$

The above expression limits the usable modulation bandwidth for a given DAC sampling rate to half. For 5GS/s, the waveform data throughput will be:

SR_{BB} = 5GHz / 8 = 625MS/s, DT = 625MS/s x 8 bytes / complex sample = 5GBps

So, 5GS/s is the maximum DAC sample rate that can be set when using the IQ Mode TWO. In the IQ mode HALF (Figure 1.4d), two channels are used together. In the Proteus units, channels are grouped in pairs sitting in the same DAC Chip. These two DACs can be grouped to act as a single channel. The I component is fed into the odd numbered channel N, while the Q component is fed into the even numbered channel N+1. The NCOs in each channel are synchronized so they work as a single quadrature NCO. Finally, the output of the even-numbered channel is disabled, and the numerical output of the Q multiplier is routed internally to the adder in DUC #1 in the odd-numbered channel. This strategy results in the availability of twice the data throughput than in IQ Mode ONE. This allows for the selection of a lower interpolation factor. For the P9484M, at 9GS/s, the baseband sample rate using the 4X interpolation factor will be

SR_{BB} = 9GHz / 4 = 2.25GS/s

While data throughput (per channel) will be

$$DT = SR_{BB} \text{ x } 1/2 \text{ IQ pair x 2 samples x 2 bytes} = SR_{BB} \text{ x } 2 \qquad (6)$$

DT = 2.25GS/s x 1/2 IQ pair x 2 samples x 2 bytes = 4.5GBps < 5GBps

This results in a modulation BW larger than 2GHz at the expense of losing half of the channels.

1.1 Frequently Asked Questions

- SCLK limits depending on IF and DUC mode: Table 1.1 Maximum DAC Sampling Rate and Modulation Bandwidth vs. DUC Mode and Interpolation Factor
- Command order for DUC programming: 2 Programming the DUC in Proteus
- IQ data formatting: 2.1 Data Formatting and Downloading for the DUC

2 Programming the DUC in Proteus

The default state of the Proteus AWG (the one after starting it up or after resetting it with the *RST) is the DIRECT conversion mode. In this mode, samples bypass the DUC block and are fed directly to the DAC. The DUC mode can only be selected when the instrument is in the 16-bit mode. 16-bit mode is the default mode when the DAC sampling rate is lower or equal than 2.5GS/s. This is always the case for the P258X Proetus models, and this is the mode for the P948X after reset as the default sampling rate for all Proteus models is 1GS/s. However, the P948x models transition to the 8-bit mode (where samples are made by 8-bit integers) when sampling rate is set to be higher than 2.5GS/s and no interpolation is applied to the incoming waveform from the memory. In practical terms, this means that sampling rate must be set to the higher than 2.5GS/s state AFTER the DUC mode has been selected and the corresponding interpolation factor is applied. The pseudo-code to set up the DUC mode in the IQ Mode ONE would be as follows:

```
:INSTrument:CHANnel 1              % Default is 1. Channel can be 1, 2, 3, or 4

% Download IQ1 interleaved waveform here

:FREQuency:RASTer 2.5E9            % Between 2.0E9 and 2.5E9. Not required after *RST
:INTerpolation X8                  % X8 is the default, alternatively use X4, X2
:MODE DUC                          % Default mode is DIRect
:IQModulation ONE                  % ONE is default
:FREQuency:RASTer 9.0E9            % Any compatible DAC sampling rate can be set now

% IQ1 interleaved waveform can be downloaded here as well

:NCO:SIXDb1 ON                     % This will increase NCO amplitude by 6dB
:NCO:CFRequency1 1.8E9             % 1.8GHz. It can be set from 0.0 up to 9.0E9
:NCO:PHASe1 45.0                   % 0.0 is the default. It can be any angle in degrees
:SOURce:VOLT 0.5                   % Output amplitude in Volts
:FUNCtion:MODE:SEGMent 1           % Segment #1 is used for generation as an example
:OUTPut ON                         % Output for the selected channel is activated
```

The above sequence of commands would result in channel 1 generating an IQ modulated signal with 1.8GHz carrier frequency at 9GS/s sample rate for the DAC. Sampling frate for the baseband signals would be 1.125GS/s so the available modulation bandwidth would be slightly larger than 1GHz. The command sequence for the TWO mode would be as follows:

```
:INSTrument:CHANnel 1              % Default is 1. Channel can be 1, 2, 3, or 4

% Download IQ1/IQ2 double interleaved waveform here

:FREQuency:RASTer 2.5E9            % Between 2.0E9 and 2.5E9. Not required after *RST
:INTerpolation X8                  % X8 is the default, alternatively use X4, X2
:MODE DUC                          % Default mode is DIRect
:IQModulation TWO                  % ONE is default
:FREQuency:RASTer 5.0E9            % Any compatible DAC sampling rate can be set now

% IQ1/IQ2 double interleaved waveform can be downloaded here as well

:NCO:SIXDb1 ON                     % This will increase NCO1 amplitude by 6dB
:NCO:CFRequency1 1.8E9             % NCO1 set to 1.8GHz. It can be set from 0.0 up to
                                   % 5.0E9
:NCO:PHASe1 45.0                   % NCO1 Phase. 0.0 default. It can be any angle
                                   % in degrees
:NCO:SIXDb2 ON                     % This will increase NCO2 amplitude by 6dB
```

```
:NCO:CFRequency2 500E6          % NCO2 set to 500MHz. It can be set from 0.0 up to
                                % 5.0E9
:NCO:PHASe2 90.0                % NCO2 Phase. 0.0 default. It can be any angle in
                                % degrees
:SOURce:VOLT 0.5                % Output amplitude in Volts
:FUNCtion:MODE:SEGMent 1        % Segment #1 is used for generation as an example
:OUTPut ON                      % Output for the selected channel is activated
```

The TWO mode limits the maximum sampling rate for the X8 interpolation factor to 5GS/s as waveform transfers are now made of 4-tuples of samples (I1/Q1/I2/Q2) doubling the data throughput from the waveform memory. Relative amplitude for the 500MHz and 1.8GHz modulated signals is controlled through the IQ values for each pair. The above sequence of commands would result in channel 1 generating one IQ modulated signal with 1.8GHz carrier frequency and another at 500MHz carrier frequency at 5GS/s sample rate for the DAC. Sampling frate for the baseband signals would be 562.5MS/s so the available modulation bandwidth would be around 500MHz.

The HALF mode handles the I and Q waveforms as independent entities. I and Q components are then downloaded independently to each participating channel in the pair. This is the pseudo-code for this mode:

```
:INSTrument:CHANnel 1           % Channel 1 will be used for I

% Download I waveform here to some segment, i.e. segment #1

:FREQuency:RASTer 2.5E9         % Between 2.0E9 and 2.5E9. Not required after *RST
:INTerpolation X4               % X8 is the default, alternatively use X4, X2
:MODE DUC                       % Default mode is DIRect
:IQModulation HALF              % ONE is default
:FREQuency:RASTer 9.0E9         % Any compatible DAC sampling rate can be set now
:NCO:SIXDb1 ON                  % This will increase NCO1 amplitude by 6dB
:NCO:CFRequency1 1.8E9          % NCO1 set to 1.8GHz. It can be set from 0.0 up to
                                % 5.0E9
:NCO:PHASe1 45.0                % NCO1 Phase. 0.0 default. It can be any angle in
                                % degrees
:FUNCtion:MODE:SEGMent 1        % Segment #1 is used for I generation as an example
:INSTrument:CHANnel 2           % Channel 2 will be used for Q

% Download Q waveform here to a different segment, i.e. segment #2

:NCO:SIXDb1 ON                  % This will increase NCO1 for Q amplitude by 6dB
:NCO:CFRequency1 1.8E9          % NCO1 for Q must be set to the same I frequency
:NCO:PHASe1 45.0                % NCO1 Phase for Q must be set to the same I phase
:FUNCtion:MODE:SEGMent 2        % Segment #2 is used for Q generation as an example
:INSTrument:CHANnel 1           % Select Channel 1 as it will be the active output
:SOURce:VOLT 0.5                % Output amplitude in Volts
:OUTPut ON                      % Output for the selected channel is activated
```

The above sequence of commands would result in channel 1 generating one IQ modulated signal with 1.8GHz carrier frequency 9GS/s sample rate for the DAC. Sampling frate for the baseband signals would be 2.25GS/s so the available modulation bandwidth would be larger than 2.1GHz. Segments for I and Q must be different as the same waveform memory bank is shared between any pair of channels in the same DAC chip (channels numbered 2N -1 and 2N, N = 1, 2).

2.1 Data Formatting and Downloading for the DUC

Complex waveform data must be properly formatted before downloading it. The best way to think about waveform data for Proteus is as a vector of unsigned 16-bit integers. For direct generation (where just a real waveform is involved), just downloading this vector to the target segment in the waveform memory will be sufficient. For the DUC mode, a complex IQ waveform is involved. In the IQ mode ONE there is a single IQ pair, see figure 2.1a below.

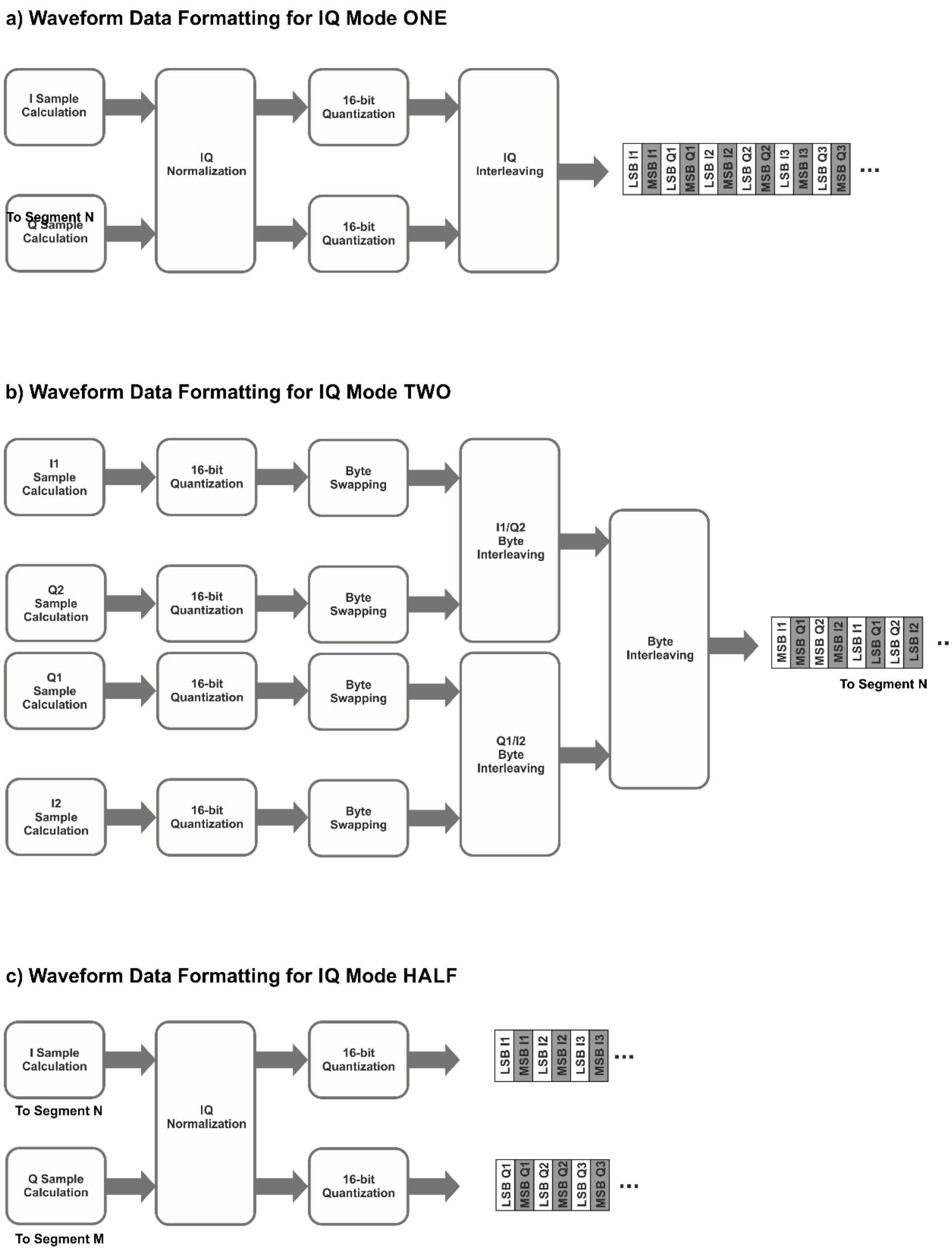


Figure 2.1 IQ waveform data must be properly multiplexed and formatted to be downloaded to the waveform memory. The format of the data depends on the IQ Mode. For the ONE mode, the 16-bit I and Q samples are just interleaved (a). In the TWO mode, there is multi-layer byte, I/Q, and IQ1/IQ2 pair interleaving process (b). The HALF mode handles I and Q waveforms as non-interleaved, separate waveforms as they go to different segments within the same memory bank (c).

The IQ pair values may be originally stored as two vectors with the same length, one for the I component, and another one for the Q component. However, for the ONE mode, a single vector (IQ) made by the interleaved elements of the I and Q vectors must be created for download using the following interleaving scheme:

```
IQ(2n - 1) = I(n);     n = 1….Size(I) = Size(Q)          (7)

IQ(2n) = Q(n);         n = 1….Size(I) = Size(Q)          (8)
```

Segments in Proteus are always defined through the number of samples. This means that for the ONE IQ Mode, segments will have twice the length of any of the components. It is also important to notice that the granularity for the complex waveforms (or any of its components) will become half that of the Proteus waveform granularity. If regular granularity is 32, it will become 16 for the complex waveforms as each complex sample fills two real sample memory positions.
The TWO IQ Mode requires a more complex data interleaving as two IQ pairs must be stored in a given segment of the waveform memory, Figure 2.1b. Given the way the data is processed internally in the Proteus unit, if we take I1, Q1, I2, and Q2 as the four vectors containing the two IQ pairs, formatting the data in a single vector IQ for download requires the following steps, as shown in figure 2.1b:

- Arrange the 16-bit samples in the I1, Q1, Q2, I2 sequence
- Split all the 16-bit samples in two bytes
- For each group of four samples, take the MSB bytes following the interleaving sequence shown above resulting in the IQMSB vector
- The same operation must be performed for the LSB bytes resulting in the IQLSB vector
- The final waveform data is obtained by interleaving the IQMSB and IQLSB vectors built in the previous steps

Notice that the final vector will be made of 8-bit unsigned integers, so its size will be 8 times the number of complex samples for each IQ pair. The resulting vector will be composed by 4 times the number of complex samples in a single IQ pair, and this will be the segment length. Again, the granularity value that must be applied to the waveform length calculations must be the regular one divided by four. This results in an actual granularity of 32/4 = 8 samples for the standard Proteus.
In the HALF IQ Mode, I and Q waveforms are handled as independent waveforms, so they do not need special formatting procedure, and must be downloaded to segments with different number as the two channels involved in the generation of the signal share the same waveform memory bank.
Downloading binary data to Proteus must be done according to the IEEE-488.2 binary block transfers standard. This model is based in transfers of 8-bit bytes. The ONE and TWO IQ Modes formatting procedures described above result in a vector made of 16-bit unsigned integers. For those modes, depending on the programming language and the communication API used, it is possible to split the 16-bit integers in two 8-bit integers before downloading or just use a binary transfer function in the library capable of doing the same internally, so a 16-bit integer vector is supplied as an input parameter. The binary block format defined by the IEEE-488.2 standard (and incorporated to the SCPI standard) defines a header for the binary block with the ***#nmm..m*** ASCII characters before the actual binary data. The **n** is a single digit (`0','1',..'9' in ASCII) expressing the number of ASCII digits (the ***m*** in the header) expressing the number of bytes in the current transfer. As an example, the header ***#41024*** would indicate that the transfer is composed by 1024 bytes that should follow immediately. Therefore, a single binary transfer

can consist in a maximum of 999,999,999 bytes (as the ***n*** is always a single digit so ***mm..m*** can be made of up to 9 digits). Proteus supports even longer segments so waveform downloads must be split between multiple transfers by using the offset mechanism supported by the Proteus platform. Even when the offset mechanism for waveform download is used, binary transfers can be segmented by calling the same function to transfer chunks of equal or different length. Using this strategy allows to keep smaller arrays in the computer memory (so much slower virtual memory usage can be limited or avoided) and also limits the time the control SW is waiting for the transfer to finish and allows for setting a lower time-out value to detect when communication stalls for some unexpected reason. Given the overhead for each transfer function call, chunks must be long enough, so overhead is not significative, but not too long, so array size and transfer time for each chunk stays reasonable.
Proteus can be accessed through any VISA compatible API or using the DLL supplied with the instrument FW. The VISA library is very well known and it is independent of the OS, communication interface, and instrument, so it is largely used in T&M control SW. However, this level of compatibility pays some price in terms of transfer speed. The alternative Tabor-supplied DLL supports direct access to the PCIe bus within the PXIe bus and provides a much higher transfer speed for large binary blocks, with a much lower overhead. This DLL works under Windows 10/11 in embedded PXI computers (i.e. the embedded computers in the Benchtop and Desktop Proteus units) or with external computers using a PCIe/PXIe bus extender implemented using MXI or Thunderbolt-based bridges.

2.2 Baseband Waveform Calculation for the Proteus DUC

Waveforms must be properly calculated in order to produce good-quality signals. These are the steps to calculate properly complex (IQ) waveforms to be used with the Proteus DUC:

1. Waveform parameter calculation so the right sample rate for the DAC and interpolation factor are selected according to the expected carrier frequency and modulation bandwidth required by the application.
2. Waveform calculation including waveform length selection, pulsed or continuous RF generation, single or multiple modulated waveforms, etc.
3. Waveform normalization to maximize SNR and signal power depending on the IQ Mode.
4. Waveform Quantization.

In order to simplify the understanding of the concepts involved, an example will be used. This example consists in the generation of a multi-tone signal with arbitrary frequency and amplitude settings within a 1000MHz band around 3GHz, and a 1GHz minimum distance to the image in the second Nyquist Zone. A MATLAB script will be used to better define all the procedures (Appendix 1).

Waveform Parameter Calculation

This step will result in the selection of the DAC sampling rate, the IQ mode and the interpolation factor to be applied in the DDC, and the settings for the NCO. The Maximum Frequency (MF) component to be generated in this example will be determined by the Carrier Frequency (CF) and the worst-case Modulation Bandwidth (MB), and the minimum distance to the unwanted images (MDI):

```
MF = CF + MB / 2 + MDI / 2                (9)
```

Using the parameters for the example

MF = 3GHz + 1GHz/2 + 1GHz/2 = 4.5GHz

The above Maximum Frequency can be implemented within the first Nyquist Zone for DAC sampling rates (SR) equal or larger to 9GS/s. Selecting 9GS/s will result in a better-quality signal with enough distance to the image in the second Nyquist Zone (1GHz) to allow for the use of a simple, inexpensive Low-Pass filter to remove it, if necessary. The second step is selecting the interpolation factor. At 9GS/s and IQ Mode ONE, the only available interpolation factor (IF) for the Proteus P948X series is 8X. Modulation Bandwidth (MB), which is equal to the Baseband Sample Rate (BBSR), for the above settings will be

$$MB = SR_{BB} = SR_{DAC} / IF \qquad (10)$$

Again, using the values defined for the example

MB = 9GHz / 8 = 1.125GHz

In fact, the actual Modulation Bandwidth is slightly lower because the roll-off of the interpolation filter. Interpolation filters for Proteus have a 0.01dB flatness for 80% of the Nyquist frequency of the input waveform (before interpolation) and usable bandwidth (-3dB) is close to 90% so the desired 1GHz modulation bandwidth is feasible with this interpolation factor. In this example, two tones must be generated with a 500MHz maximum distance to the carrier frequency. If a larger than 1GHz distance between tones must be implemented, there are two ways of doing it using the DUC block:

1. Use of the HALF IQ Mode: In this mode, modulation bandwidth around the central frequency when the DAC sampling rate is 9GS/s will be 2.25GHz when setting the interpolation factor to 4X. In this case, just half of the output channels will be available.
2. Use of the TWO IQ Mode: In this mode, two DUCs are used for the same channel. To generate two tones, it is not even necessary to apply any complex rotation to shift the position of the tone respect to the carrier frequency. Instead, just setting each NCO to the final frequency and apply an "all 1s" to one of the Components will result in two tones at any frequency between DC and SR/2.

The first step is selecting the waveform length so the intended multi-tone signal can be implemented. One way to start defining the required waveform length is by defining a "frequency resolution" parameter (FR). To be always able to synthesize any tone defined with the specified frequency resolution while keeping phase continuity, a time window (TW) equal to the inverse of this FR parameter must be implemented (or any integer multiple of it):

$$TW = K / FR, \; K = 1, 2, \ldots N \qquad (11)$$

Waveform Length (WL) can be calculated now:

$$WL = TW \times SR_{BB} \qquad (12)$$

If 1MHz is selected as the FR parameter

TW = K / 1MHz = K μs

If SR is 9GS/s, and IF is 8X

SR_{BB} = 9GS/s / 8 = 1.125GS/s
WL = K x 1E-6 x 1.125E9 = K x 1125 samples

This is the raw waveform length. However, depending on the frequency of the tones, WL can be further reduced. Depending on the number of cycles, the same exact sequence of samples may be exactly repeated several times for each tone. If a common repetition period is found, the waveform length can be reduced to this period. The way to calculate this is by finding the greatest common divider (GCD) between the WL, and the number of cycles (NC) for all the tones (always an integer number when tone frequencies, TFR, are rounded to the nearest multiple of FR):

```
NC(i) = abs((TFR(i) – FC) * TW)          (13)

GCD = gcd(WL, NC(1),…(NC(N))             (14)

WL’ = WL / GCD                           (15)
```

As an example, if CF = 3GHz, TFR(1) = 2.9GHz, and TFR(2) = 3.3GHz, and K = 1

NC(1) = 100, NC(2) = 300
GCD = gcd(1125, 100, 300) = 25
WL’ = 1125 / 25 = 45 samples

The above WL’ is not the final one as this number does not meet the requirements for a waveform to be generated by the Proteus AWG. Proteus require the waveform length of waveforms stored in the waveform memory to be a multiple of 32, its basic granularity (BG). As each complex waveforms are made of two real samples (I and Q), in the IQ mode ONE complex waveform must have an actual granularity (AG) of 32/2 = 16 samples. The easiest way to meet this condition is by storing multiple repetitions of the basic waveform until the overall number of samples is a multiple of the AG parameter. The waveform length can be expressed as the Least Common Multiple (LCM) of the waveform length (WL’) and the actual granularity (AG)

```
WL’’ = lcm(WL’, AG)                      (16)
```

For the example being calculated

WL’’ = lcm(45, 16) = 720 samples

The same basic waveform will be repeated 16 times, in this case. This is not the only way to adjust the modulating signal to the waveform memory. An interesting alternative is using the closest lower multiple of the actual granularity to the basic waveform length. Following the same example

SR_{BB} = 9GS/s / 8 = 1.125GS/s

WL = K x 1E-6 x 1.125E9 = K x 1125 samples, K = 1 -> WL = 1125 samples

As 1125 is not a multiple of the Actual Granularity (AG) parameter, 16 in this case, waveform length must be adjusted

```
WL’ = floor (WL / AG) x AG               (17)
```

For this example, it will result in

WL' = floor(1125 / 16) x 16 = 70 x 16 = 1120 samples

Unless WL' / AG is divisible by AG again, there is no way to apply the previously shown waveform length optimization procedure. Often, this method results in lower waveform lengths, but not in this case. There is an important issue, though. In order to keep the right timing and frequencies for the output waveform, the final baseband (and DAC) sampling rate must be changed so the same time window is preserved. In this case

$$WL/(SR_{DAC}/IF) = WL'/(SR_{DAC}'/IF) = TW, \quad SR_{DAC}' = SR_{DAC} \times WL'/WL \quad (18)$$

In this case,

SR_{DAC}' = 9GS/s x 1120 / 1125 = 8.96GS/s

Some applications may require multiple segments with different requirements. Using the late methodology may result in different effective sample rates (SR') what may not be possible to apply because they must be generated by different channels in the same module or in the same sequence in one or more channels. Using the same sample rate for different channels or segments when it should be different will result in timing and frequency errors that not all the applications can withstand.

IQ Waveforms Normalization

Waveforms to be used for direct generation (no DUC involved) are quite straightforward to normalize. Using as much of the available DAC range as possible will result in the highest amplitude and the best SFDR signal. Waveform calculations may result in any numeric range and most times they consist in a vector of floating-point numbers. It may be useful to normalize to map the range of the vector to some more convenient range. A very popular range for normalization is -1.0/+1.0 where -1.0 is aligned with the lowest DAC output level while the +1.0 is aligned with the highest DAC output level. There are two canonical ways to map the input range for the calculated waveform to the -1.0/+1.0 range:

1. Mapping the highest value in the incoming waveform vector to the +1.0 value and the lowest value to the -1.0 value. In this way, the full range of the DAC will be used. However, the relative DC level of the signal may be not preserved.
2. Mapping the highest absolute amplitude value to +1.0 and the 0 level to 0.0. Using this mapping the maximum DAC range preserving the relative DC level will be used. In this case, unless the input waveform is symmetrical around the 0.0 level, not all the DAC range will be used.

Method 1 does not preserve the DC level of the incoming vector. However, AWGs in direct conversion (no DUC) can compensate for this using the DC Offset control. Anyway, this method cannot be used to normalize complex IQ waveforms as each one of the components must preserve the right DC level (typically 0) to avoid carrier leakage.

Another constraint is related with the quadrature modulator functionality. The NCO numerical IQ output and the associated multipliers are designed in such a way that if just one of the components is being applied to the modulator and it is using the full integer range at the input (16-bit for Proteus), the numerical output of the IQ modulator will use the full DAC range without suffering any clipping effect. When both components, I and Q, are being fed to the IQ modulator, there is a chance that the output of the adder after the multipliers go beyond the lower and upper limits of the DAC, resulting in an extremely non-linear clipped signal. The best way to avoid this effect is by normalizing both components so the maximum module of the I/Q pair in the complex waveform is mapped to +1.0 normalized level and the 0 level is mapped to the 0.0 level, see figure 2.2a below.

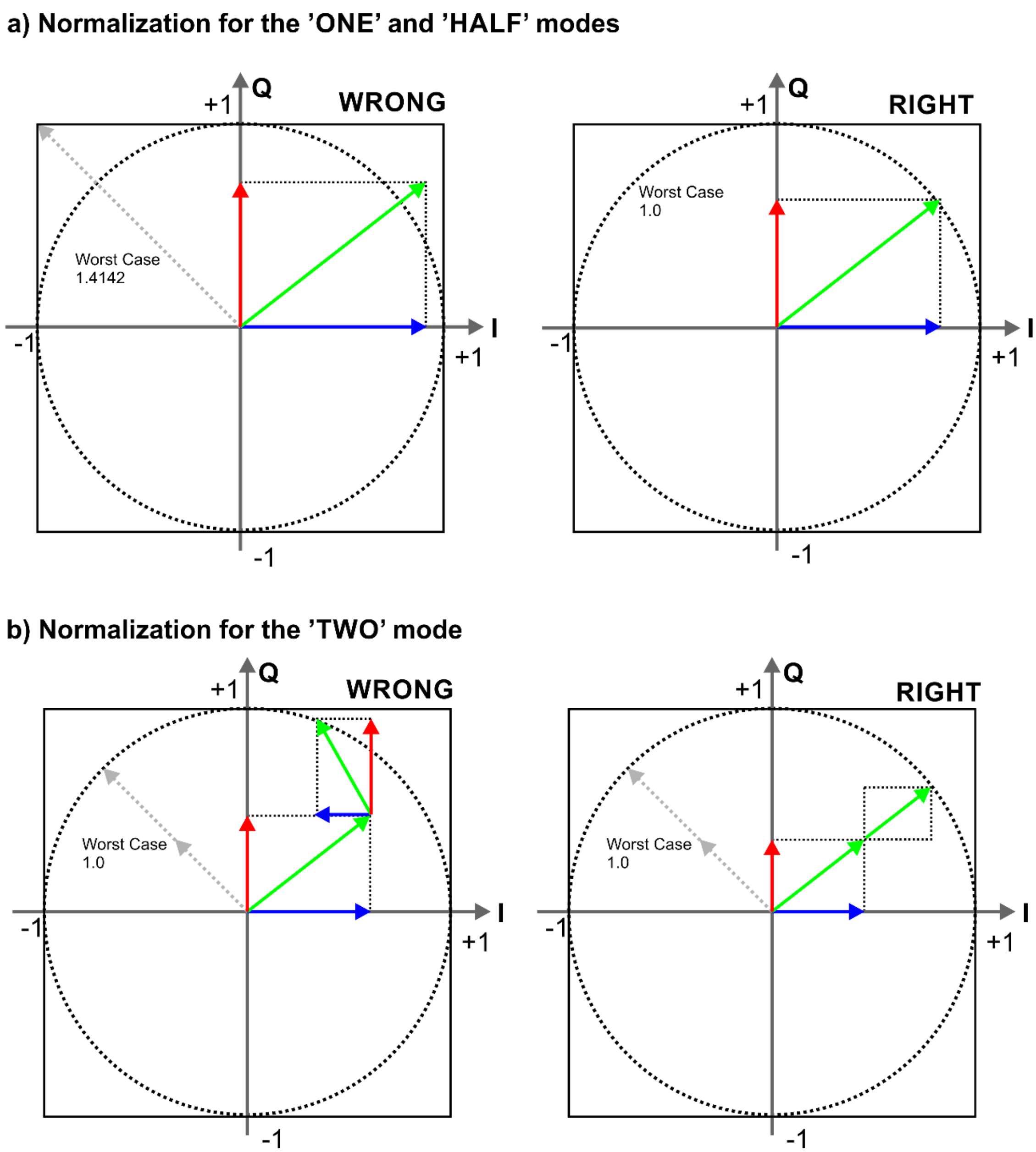


Figure 2.2 Normalization of I and Q waveforms must be performed prior to quantization to make sure the DUC will not clip. Just normalizing I and Q independently will not guarantee unclipped signals. For the ONE and HALF mode, normalization is performed by making sure the module of the complex signal (I + jQ) is always lower than 1.0 (a). In the TWO mode, at any sample time, the IQ1 and IQ2 waveforms combine depending on the instantaneous phase of the corresponding carriers. However, as NCOs run independently, worst case scenario (when both phases are aligned) may happen at any moment so normalization must make sure that the worst case combination of modules for IQ1 and IQ2 is lower than 1.0 (b).

In this way, DC levels I and Q relative levels will be preserved, while clipping will be avoided. So, if IWFM and QWFM are the non-normalized waveforms, the Normalization Factor (NF) must be calculated, and the normalized ones will be expressed by

$$NF = max((IWFM^2 + QWFM^2)^{1/2}) \qquad (19)$$

$$IWFM' = IWFM / NF, \; QWFM' = QWFM / NF \qquad (20)$$

The above expression can be used to normalize IQ waveforms when Proteus uses the IQ ONE or HALF modes. For the TWO mode, things are more complex as two different IQ pairs must be generated. The only way to avoid any problem is looking for the worst-case module of the combined waveform, see Figure 2.2. The worst-case scenario can be found by thinking that NCOs work coherently at the same frequency. Expression (16) above can be modified accordingly

$$NF = max((IWFM1^2 + QWFM1^2)^{1/2}+(IWFM2^2 + QWFM2^2)^{1/2}) \qquad (21)$$

$$IWFM1' = IWFM1 / NF, \; QWFM1' = QWFM1 / NF$$

$$IWFM2' = IWFM2 / NF, \; QWFM2' = QWFM2 / NF \qquad (22)$$

2.2.1 Interpolation-related Clipping

Even when applying proper normalization, clipping may happen as the interpolation process can reach even more extreme values, especially for narrow peaks, quite typical in multi-tone and OFDM signals with high bandwidths relative to SR_{BB}. In this case, interpolated samples my go out of the DAC range (top and bottom) and generate clipping resulting in lower-than-expected PAPR, spectral growth, and reduced EVM performance. Although, interpolated waveforms could be simulated so the absolute peak values could be found and the normalization factor corrected, this is a calculation intensive procedure that could be avoided by forcing the final normalization factor to a higher value than the one calculated with the methodology described previously. Ideally, correcting the normalization factor without calculating the full interpolated waveform should be done through the analysis of the worst-case scenario. Worst-case scenario can be easily calculated if the interpolation filter in known. It requires a specific sequence of input samples that when convolved to the filter results in the highest possible positive or negative peaks. For a general input sample sequence (bound to the -1.0/+1.0 range) and a symmetrical filter (interpolation filters are always symmetrical to obtain linear phase response), it is quite easy to find out that this sequence of samples is:

$$X(n) = sign(H(n)) \qquad (23)$$

$$NF = sum(abs(H(n))) \qquad (24)$$

However, for interpolators, the sequence of input samples consists in the input samples every IF samples and IF-1 zeros in the middle. This is equivalent to using IF filters in parallel where the input samples are applied at the original sample rate, so the interpolated waveform is obtained by multiplexing the output of all the filters. Each one of the sub-filters consists in taking one of every IF samples from the overall interpolation filters by shifting the initial sample by one for each filter. The problem now is finding which one of the sub-filters results in the highest peak:

$$H_K(n) = H(IF * n + k),\ k = 0,..,IF-1 \quad (25)$$

$$NF_K = sum(abs(H_K(n))) \quad (26)$$

$$NF = max(NF_1,\dots,\ NF_{IF-1}) \quad (27)$$

If we apply the above expressions to the actual interpolation filter used for 8x interpolation in the Proteus family of products, NF is 2.3157 for a normalized waveform in the -1.0/+1.0 range. In other words, in order to be absolutely free of interpolation-related clipping, no matter the input samples, the normalized waveform must be attenuated by 7.3dB. This normalization factor correction may be unacceptable as the peak power and SNR will be reduced by the same factor.

A less conservative approach may be taking the peak reached in the worst-case transition (-1.0 to 1.0) in one sample time as the reference. When this transition is applied to the Proteus 8x interpolation filter, the interpolated signals shown is obtained, see the figure below. NF is now 1.27483, or 2.1dB. This will result in a better than 5dB improvement respect the worst-case scenario, but it does not guarantee a clipping-free interpolation.

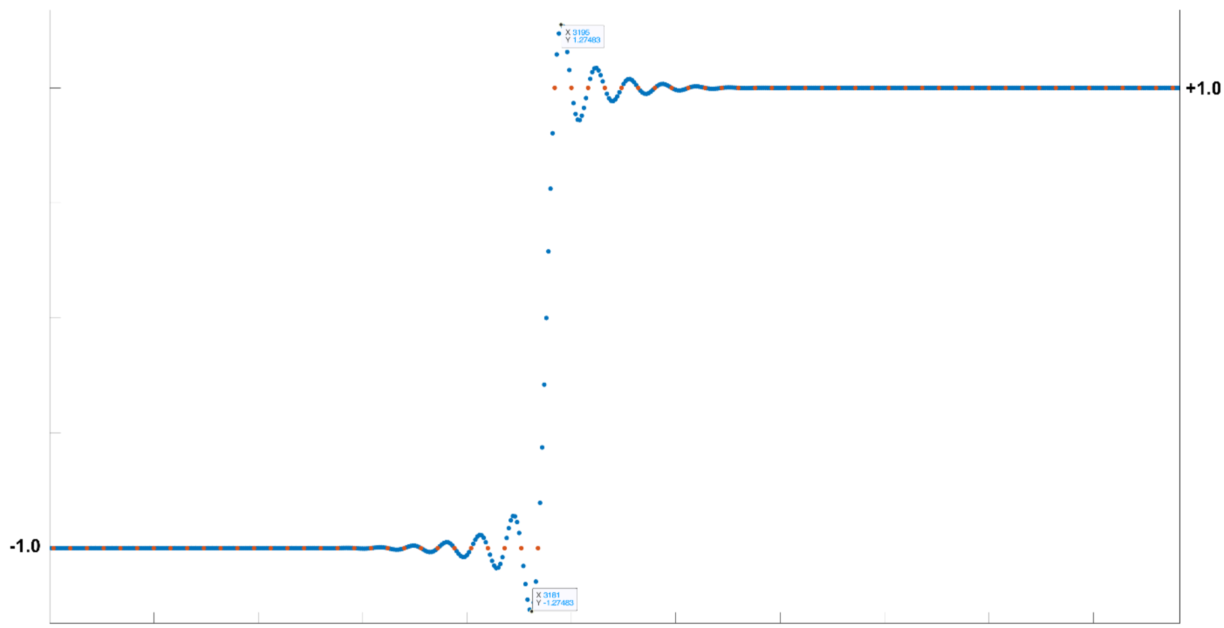


Figure 2.3 Original (red dots) and 8x interpolated (blue dots) maximum amplitude instantaneous transient using the Proteus interpolator. In this case, the absolute value of the maximum and minimum peaks is 1.27483. If interpolation related clipping must be avoided, the input normalized waveform should be divided by the same number so the absolute value of the maximum and minimum peaks are +1.0 and -1.0 again.

In order to analyze how well the above methodologies can be applied to calculating a generic normalization factor, a statistical analysis may be required. First, multiple normalized multi-tone signal with random phases will be used to analyze the normalization factors in a statistical way. The combination of the number of tones and tone-spacing are selected in such a way the full modulation BW (SR_{BB}) is used (kind of worse-case scenario). For a test with 10,000 different multi-tone signals with different random phase distributions, the maximum normalization factor obtained is 1.51 and the histogram for the distribution of values can be seen in the figure below. Selecting 1.5 (or 3.52dB) as the additional correction factor should result in no clipping for a vast majority of input waveforms.

Selecting the right interpolation-related normalization factor depends on the test situation. Typically, when the input waveform bandwidth is much lower than the modulation BW for the selected SR_{DAC} and IF, the additional NF can be very close to 1.0 (1.1 to 1.3). For wideband signals with random-like frequency-domain contents (multi-tone with random phases, OFDM baseband signals), selecting a 1.4-1.5 additional NF should result in clipping free signals at the output of the interpolator in about 99.9% of cases, and if clipping happens, the impact of it (EVM degradation, spectral growth) should be quite limited. For any situation where the best SNR and no clipping at all must be accomplished, the best solution is running a simulation of the interpolation process so the actual absolute maximum for the interpolated waveform is found, and the corresponding NF is applied.

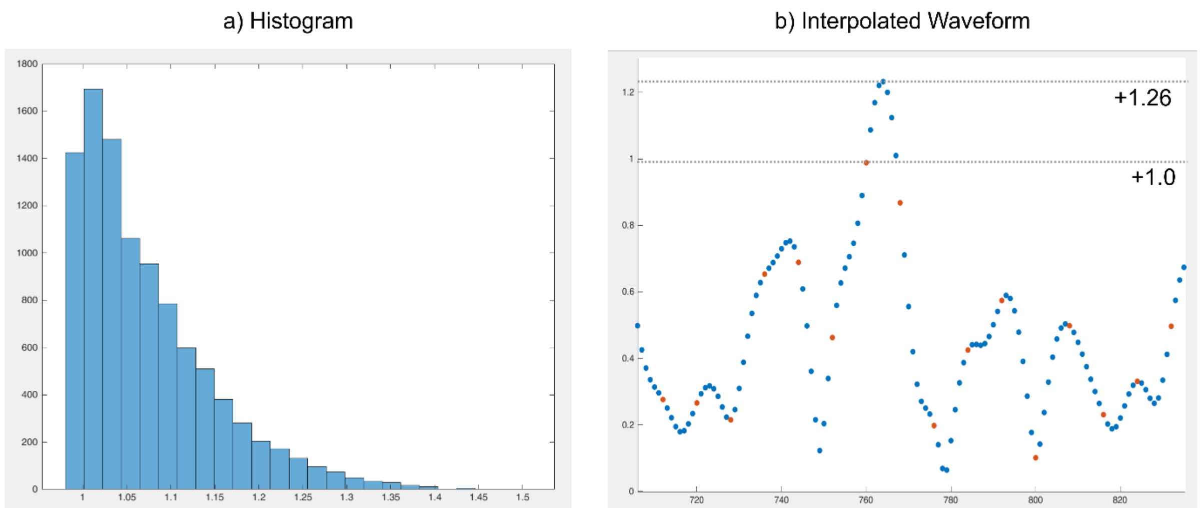
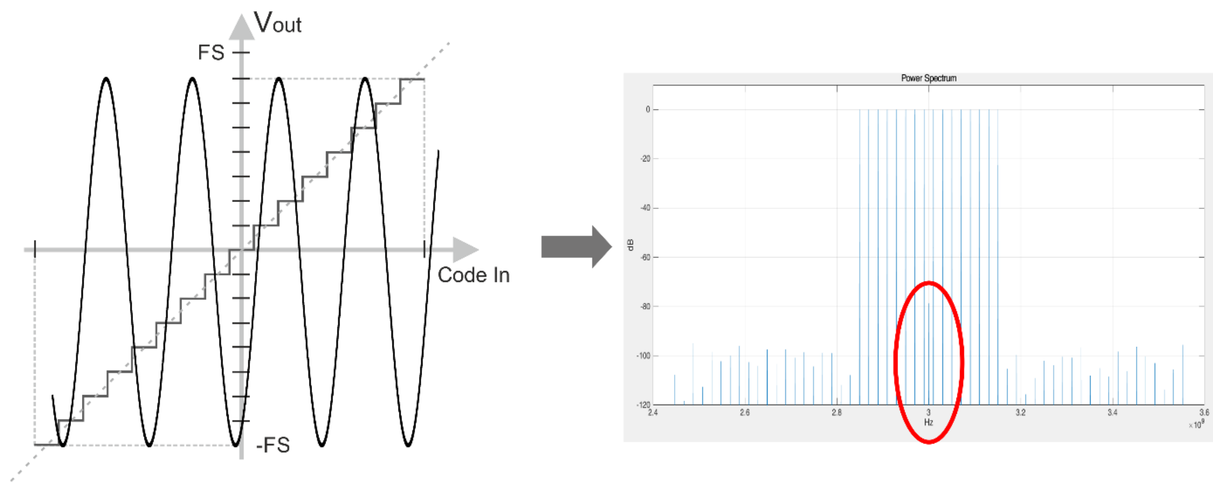


Figure 2.4 Statistical analysis of the maximum peak for a multi-tone signal with a random phase and using the full modulation BW of the DUC. After generating 10,000 different waveforms and simulating the effects of the Proteus' 8x interpolator, histogram in a) has been obtained. The maximum peak is 1.51 so selecting this additional normalization factor would result in 99.99% of waveforms without showing any interpolation-related clipping effect. In b) a detail of one of the maximum peaks in one of the acquisitions is shown. While the maximum value in the original waveform (red dots) is +1.0, the maximum in the interpolated waveform is +1.26. Less than 5% of the 10,000 waveforms in the statistical analysis go beyond 1.26 maximum peak.

2.3 IQ Waveforms Quantization

Once waveforms have been normalized, samples must be quantized to the integer size of the waveform memory. For Proteus, I/Q samples are stored as 16-bit unsigned integers. The 0.0 DC level corresponds to the 2^{16} / 2 = 32768 level. The distance between the DAC 0 level and the 0.0 DC level will be 32768, while the distance between the DAC maximum level and the 0.0 DC level will be 32767. This small asymmetry means that mapping the -1.0 to the minimum DAC level and the +1.0 to the maximum DAC level (65535) will cause a tiny DC level that will generate an small but detectable carrier leakage at the output of the IQ modulator. This can be solved by mapping the -1.0 level to the DAC level 1. In this way, symmetry is perfect and there will not be any carrier leakage at the output, see below figure.

a) 0 / 2^N - 1 DAC Range



b) 1 / 2^N - 1 DAC Range

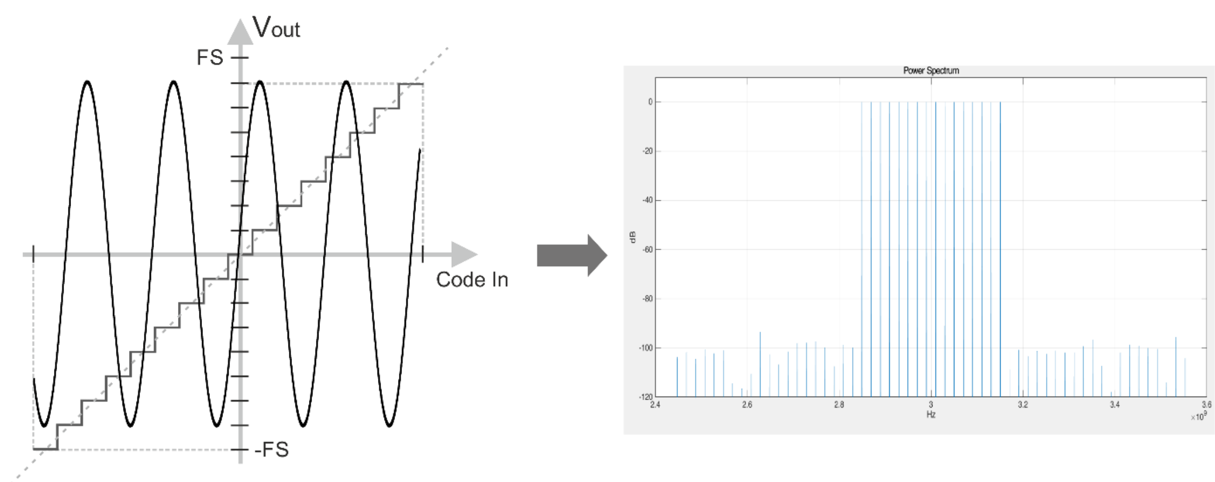


Figure 2.3 Quantization must be performed by selecting the binary level closest to the sample. It is important to map the -1.0/+1-0 range to the 1/2^N-1 range in the DAC. If the 0/2^N-1 range is used (a), a small carrier leakage will show up (-78dBc in this example). Using the right DAC range (b), the residual carrier disappears.

As an additional improvement, sample values should be rounded to the nearest quantization level after mapping the -1.0 level in the normalized waveform to the 0.5 level (so 1 – ½ LSB) and the +1.0 level to the 65535.5 (2^{16} -1 + ½ LSB). This will be equivalent to "stretch" the signal by one additional quantization level without causing any clipping.

DUCs are not forgiving when overdriving the IQ modulator as it results in hard clipping. In traditional analog IQ modulators, some overdriving may be acceptable as non-linear distortion will show up progressively. In some cases, it is possible to improve output power and SNR at the expense of some spectral growth. As DUC results in hard clipping, overdriving IQ modulator will quickly reduce signal quality, so typically, clipping should be always avoided. This means that the maximum power of the output signal depends on the DAC voltage range and the PAPR (Peak-to-Average Power Ratio) or Crest Factor of the signal being generated. This is especially critical when multiple carriers are being generated simultaneously. Multi-tone and OFDM signal generation are good examples of signals with potentially high PAPR resulting in lower power signals. It is important to optimize (reduce) PAPR as much as possible, so SNR and output power is maximized.

Generating Baseband Signals in the DUC Mode

Some applications may require RF and baseband (non-modulated) signals simultaneously. Envelope tracking is a good example. In envelope tracking amplifiers, power efficiency and working temperature are optimized controlling the power supply voltage to the amplifier so it "tracks" the envelope (instantaneous RF amplitude) of the RF signal being amplified. If just a Proteus module is available, and the DUC mode is chosen for high quality generation of the RF signal, all the channels in that module will work in the DUC mode. Fortunately, the DUC block is flexible enough to also generate a baseband signal. Any channel not being involved in the generation of a modulated or unmodulated RF signal can be used to generate a synchronous (or not) baseband signal if the following methodology is used:

- The baseband signal must be sampled at the same speed than the IQ waveforms being used for modulation in the DUC of the channels generating RF signals.
- The DUC always requires an IQ waveform so the baseband waveform must be handled as the I component and associated to an "all zeros" Q waveform. Anyway, the Q waveform will not influence the output waveform at all in this scheme.
- Once downloaded after interleaving of the IQ samples, the NCO in the DUC for the baseband channel must be set to 0.0 Hz and its phase to 0 degrees. As the I output of the NCO will be just cos(0) = 1.0 (a continuous DC level), the I signal will be interpolated and it will go through the IQ modulator unaltered. Any Q component will be multiplied by -sin(0) = 0 so it will not contribute to the output.

In 4 Appendix 1 – MATLAB Programming Example, an example "envelope tracking" script is shown. In the TWO mode, one of the DUCs can be used to generate the RF signal while the other can add a variable DC offset to the output by using it in the way described in this section. The pseudocode showing the sequence of SCPI commands follows here:

```
:INSTrument:CHANnel 1           % Default is 1. Channel can be 1, 2, 3, or 4
:FREQuency:RASTer 2.5E9         % Between 2.0E9 and 2.5E9. Not required after *RST
INTerpolation X8                % X8 is the default, alternatively use X4, X2
:MODE DUC                       % Default mode is DIRect
:IQModulation ONE               % ONE is default
% IQ1 interleaved waveform can be downloaded here to segment #1
:NCO:SIXDb1 ON                  % This will increase NCO1 amplitude by 6dB
:NCO:CFRequency1 1.8E9          % NCO1 set to 1.8GHz. It can be set from 0.0 up to
5.0E9
NCO:PHASe1 45.0                 % NCO1 Phase. 0.0 default. It can be any angle in
degrees
:SOURce:VOLT 0.5                % Output amplitude in Volts
:FUNCtion:MODE:SEGMent 1        % Segment #1 is used for generation as an example
:OUTPut ON                      % Output for the selected channel is activated
:INSTrument:CHANnel 2           % Default is 1. Channel can be 1, 2, 3, or 4
% Baseband interleaved with "all zeros" waveform can be downloaded here to segment #2
:NCO:SIXDb1 ON                  % This will increase NCO1 amplitude by 6dB
:NCO:CFRequency1 0.0E9          % NCO1 set to DC.
:NCO:PHASe1 0.0                 % NCO1 Phase is set to 0 for I samples
```

```
:SOURce:VOLT 0.5               % Output amplitude in Volts
:FUNCtion:MODE:SEGMent 2       % Segment #1 is used for generation as an example
:OUTPut ON                     % Output for the selected channel is activated
:FREQuency:RASTer 9.0E9        % Any compatible DAC sampling rate can be set now
```

2.4 Resampling

In the previous discussions, baseband data is calculated according to the previously defined baseband sample rate (SR_{BB}), which can be calculated from the DAC sample rate and the interpolation factor being used in the DUC, as shown in expression (1) above. Samples calculated in this way can be directly downloaded to the waveform memory for generation. However, in some cases, the baseband sample rate may be defined independently, or it cannot be freely selected:

1. Baseband waveform data is generated by some mathematical or application-oriented package where sample rate may be internally selected to reduce the calculation time for complex and long waveforms and/or because some sampling rate is more convenient to obtain faster, more accurate results, The lates is especially true for OFDM signals where the baseband IQ waveforms are obtained by applying the IFFT to a signal defined in the frequency domain with some specific frequency resolution, which translates automatically to a time-domain sample rate.
2. Baseband waveform data has been captured by some instruments running at their own sample rates. Examples of these include VSAs, DSOs, or RF Recorders.

Sometimes playing with the selection of the DAC sample rate and interpolation factors, it is possible to just use the already available waveform data for generation. More often, this strategy is not possible as there is no way to adapt the settings of the DUC and the DAC to the existing waveform data and obtain a valid quality signal. This does not mean that the waveform cannot be used for generation using the DUC. In fact, the only important parameter that enables the generation of any waveform with the DUC is its modulation bandwidth. If the modulation BW falls within the limits of the DUC, then the waveform can be adapted to any valid baseband sampling rate through a process called resampling. Resampling can increase (up-sampling) or reduce (down-sampling) the sample rate of the input waveform. In fact, interpolation is just one example of up-sampling.

Resampling transforms the sampling rate of a waveform without modifying the signal in the time or frequency domains. When applied to AWGs, this is equivalent to transform the waveform length while keeping the same time window. For a waveform sampled at a given sample rate (SR) with a given waveform length (WL), a new waveform length (WL’) can be calculated for any new sample rate (SR’):

```
WL’ = WL x SR / SR’                                    (28)
```

The expression above does not guarantee that WL’ will be an integer. The most practical way to handle this issue is by modifying the above expression to

```
WL’’ = floor(WL’) = round(WL x SR / SR’)               (29)
```

The time window will not be the same that the original if WL’ was not an integer. In order to keep the original time window and the same signal in both the time and the frequency domain, the actual final sample rate (SR’’) must be modified to

```
SR’’ = SR’ x WL / WL’’                                 (30)
```

Using the floor function in () will make sure that SR’’ <= SR’ as typically SR’ is the maximum sample rate for a given IQ Mode and interpolation factor, so it can be reduced but not increased.

As WL and WL'' are integers, they can be expressed as follows:

```
W/W'' = N / D                                   (31)
```

The first step is finding the non-reducible fraction equivalent to N/D

```
N' = N / gcd(N,D), D' = D / gcd(N,D)            (32)
```

The reason to reduce the fraction is to simplify some calculations. Once N' and D' are known, the next operation is performing a zero-padding with a N'-to-1 factor, see figure below.

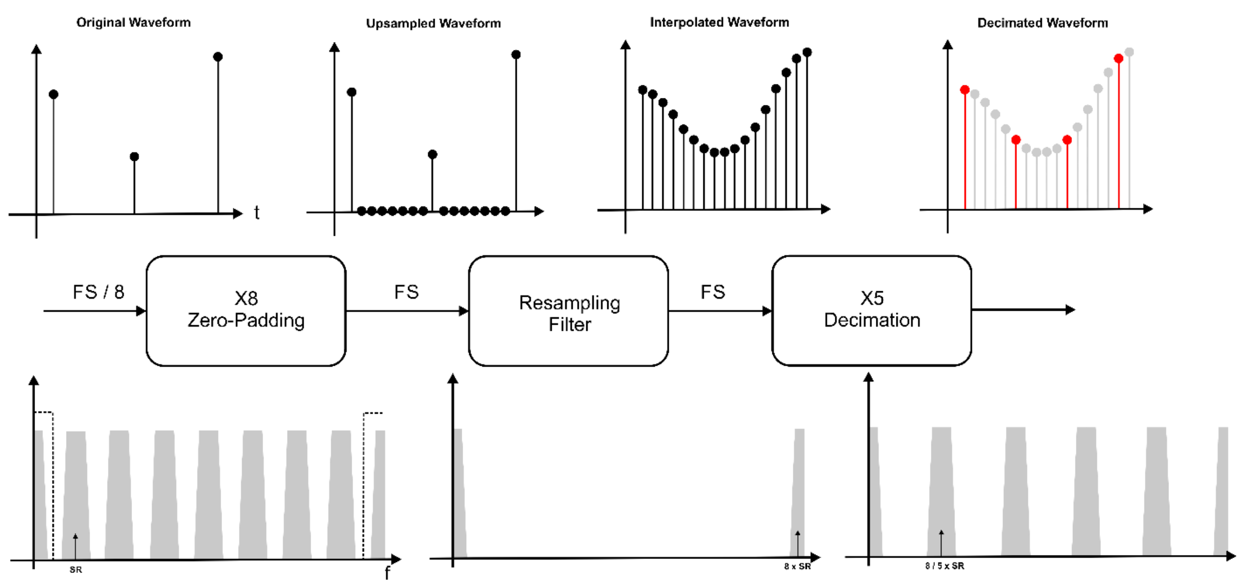


Figure 2.4 Traditional resampling algorithm are quite similar to interpolation when the input and ouput sampling rates have a N/D fractional ratio. The main difference is the resampling filter (same for upsampling and lower frequency cutoff for downsampling) and the addition of the decimation process. Here, an example of N/D = 5/8 resampling is shown. The main problem with this methodology is that it can result in huge intermediate waveforms and take a very long time to calculate.

The temporary waveform will have a sampling rate equal to N' x SR. The next step would be applying an ideal interpolation filter to obtain the intermediate samples. Then, an effective, linear-phase, low-pass filter should be applied to the interpolated signal to limit the bandwidth of the waveform to SR''/2 (the new Nyquist Frequency). If the bandwidth of the waveform is strictly lower than SR''/2, the low-pass filter may not be necessary. The interpolation filter and the low-pass filter can be combined in a single step, speeding up calculations. This combination can be called "resampling filter". Once the signal is filtered (so there is nothing beyond the SR''/2 frequency) the final SR'' frequency can be obtained by simply decimating the waveform by a D' factor (so one every D' samples are preserved).

Using the above scheme is not practical when N' and D' are big numbers and the input waveform is long. The initial up-sampling process may result in an extremely long intermediate waveform so the computer can run out of memory or be forced to use the much slower virtual memory. Even if the required intermediate data can be handled by the computer, calculation time may be unacceptable, especially when the waveform must be calculated at runtime. In 4 Appendix 1 – MATLAB Programming Example an alternative resampling algorithm is implemented as a MATLAB function. This algorithm calculates directly

the output samples without the described zero-padding, interpolation, antialiasing filter, and decimation process so there are not intermediate waveforms. A resampling filter combining the ideal interpolator and the antialiasing filter is applied and it works just the same when the sampling rate must be increased or decreased. Calculation speed for this algorithm does not depend on the N' factor.
To better illustrate this procedure, a real case will be analyzed. In this example a 2GHz BW 802.11ad OFDM signal will be generated using a Proteus P9484M AWG, see figure below.

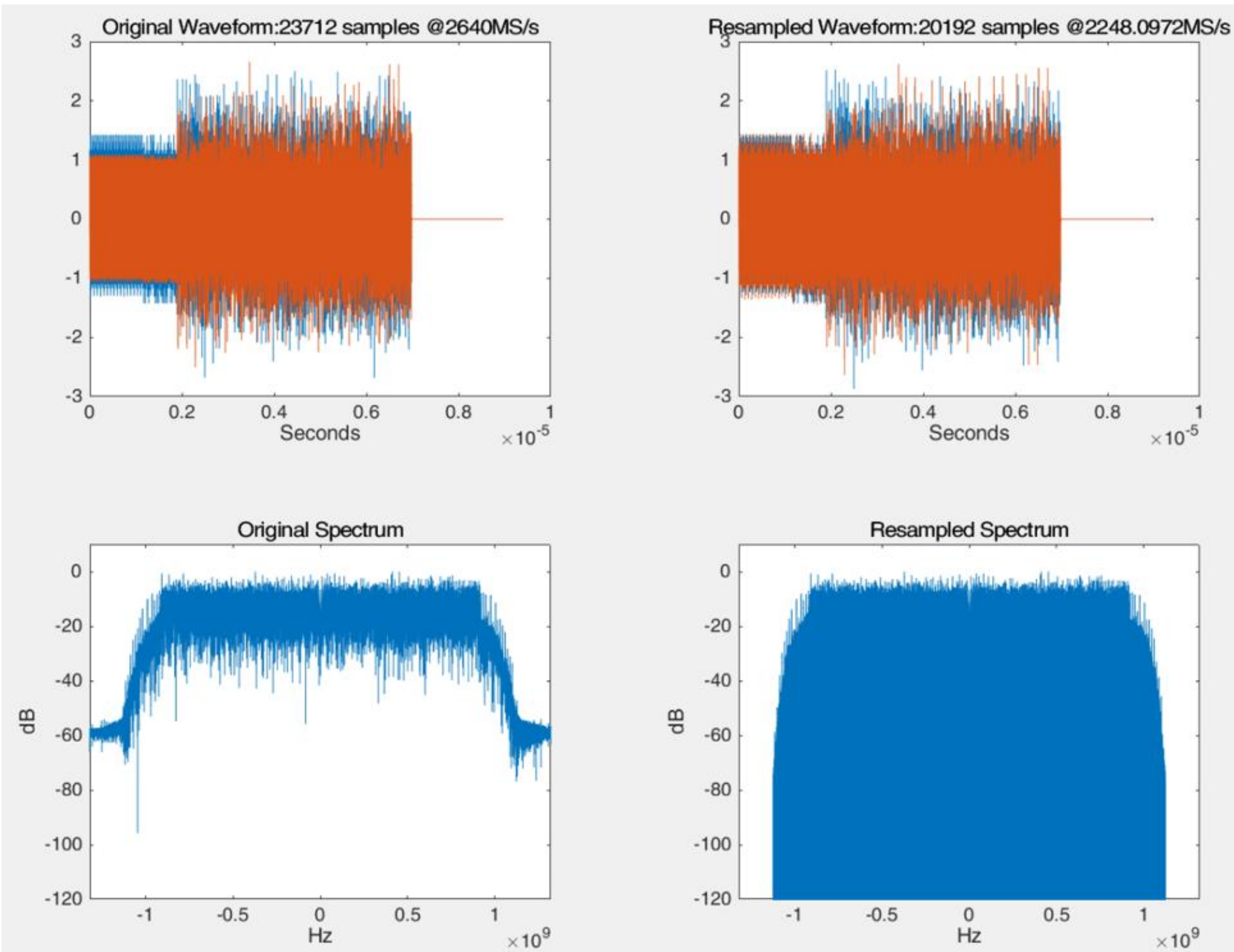


Figure 2.5 In this simulation, an 802.11ad signal calculated at the OFDM sampling rate (2640MS/s) is resampled to be generated by Proteus at 9GS/s in the DUC mode, IQ Mode HALF, and 4X interpolation. This results in a target SR_{BB} of 2.25GS/s. once the resampling algorithm is applied, all the information of the signal is preserved (almost 2GHz Bandwidth). The resulting SR_{BB} is 2248.0972MS/s after correcting it given the waveform length granularity that must be applied.

Given the modulation bandwidth involved, the highest sampling rate, 9GS/s will be selected. For this signal, the IQ mode ONE is not sufficient as modulation BW, at 9GS/s and 8X interpolation factor, is around 1GHz. Instead, the IQ mode TWO will be used. This mode enables the X4 interpolation factor, so the available modulation BW exceeds the 2GHz barrier. The target baseband sample rate will be 9GS/s / 4 = 2.25GS/s. To generate the baseband signal, the WLAN Toolbox from MATLAB will be used. Generating a RF packet with this toolbox is straightforward. The calculation process for the complex (IQ) baseband waveform data is made using the native sampling rate for the OFDM's IFFT, 2.64GS/s. As the maximum sampling rate for the IQ baseband signals in the mode TWO is 2.25GS/s, the waveform must be resampled (downsampled in this case) from 2.64GS/s down to 2.25GS/s. In this example, the input waveform (a single 802.11ad packet) from MATLAB is made of 23,712 samples. This means that the Time Window (TW) for it is

$$TW = 23{,}712 \ / \ 2.64GS/s = 8.981818\mu s$$

For waveform length granularity being 32 samples, (IQ mode TWO uses non-interleaved I/Q waveforms), to get about the same TW at 2.25GS/s, Waveform Length (WL´) must be

$$WL' = floor(23{,}712 \ x \ 2.25 \ / \ (2.64 \ x \ 32)) \ x \ 32 = 20{,}192$$

In order to get exactly the same Time Window, the 2.25GS must be corrected according to the WL'

$$SR_{BB}' = SR_{BB} \ x \ WL' \ / \ WL = 2.64GS/s \ x \ 20{,}192 \ / \ 23{,}712 = 2.2481GS/s$$

And SR_{DAC} will be

$$SR_{DAC} = SR_{BB} \ x \ IF = 2.2481 \ x \ 4 = 8.9924GS/s$$

As WL is 23,172 and WL' is 20,192, the zero-padding (N') and the decimation (D') factor will be

$$N' = 20{,}192 \ / \ 4 = 5{,}048, \ D' = 23{,}172 \ / \ 4 = 5{,}793, \ gcd(20{,}192, \ 23{,}172) = 4$$

If the traditional resampling scheme is used, the length intermediate waveform (WL'') would be

$$WL'' = WL' \ x \ N' = 20{,}192 \ x \ 5{,}048 = 101{,}929{,}216 \ samples$$

Even for a relatively short waveform, calculations may be rather long and memory requirements very high. Resampling can be a dangerous and difficult when the sample rate for the original waveform is too close to twice its bandwidth, see figure below.

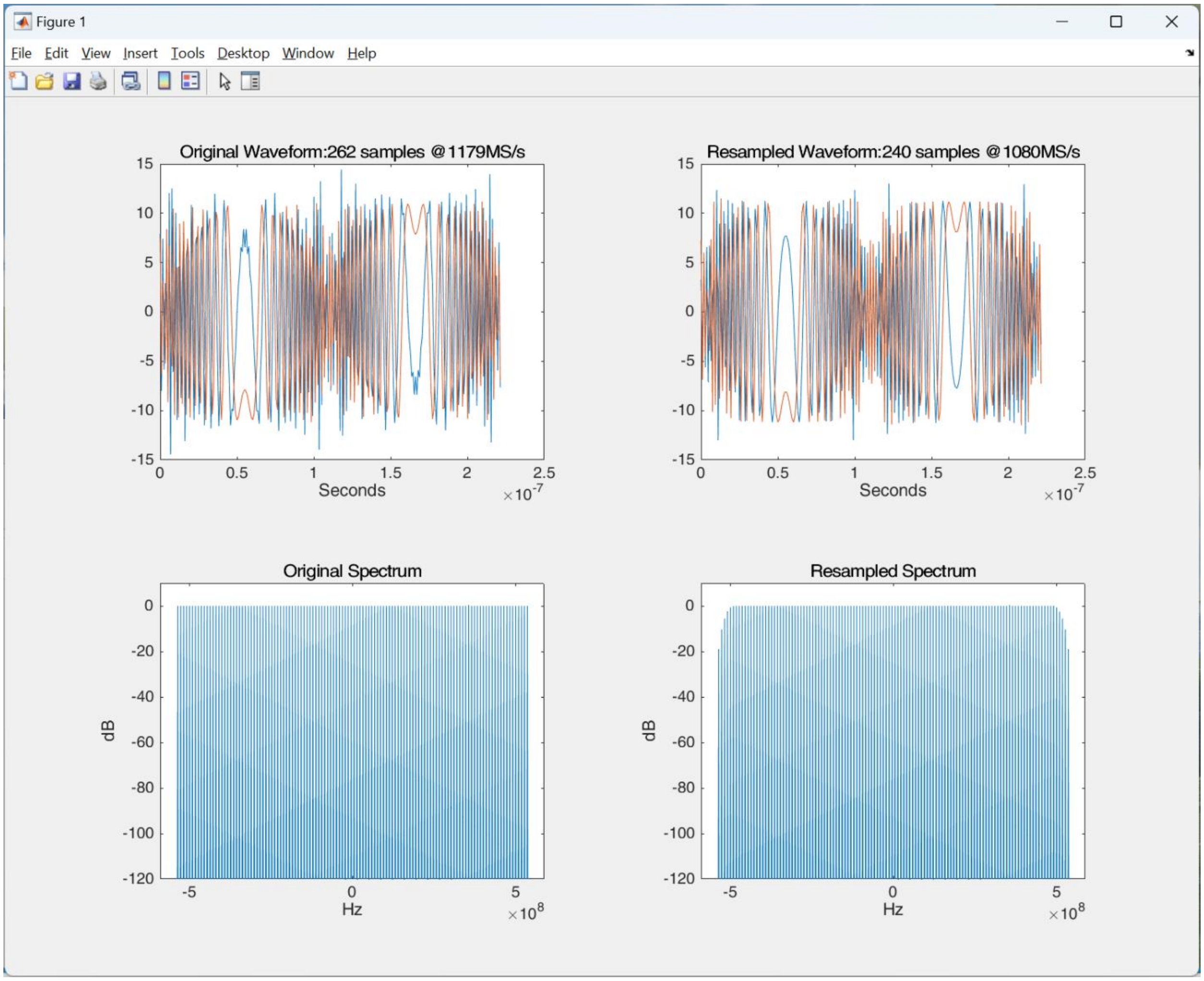


Figure 2.6 Resampling when the bandwidth of the input signal is close, equal, or higher than the final SR_{BB} will result in the linear distortion of the high frequency components and, eventually, in some image interference. Here a multi-tone signal with BW close to the target SR_{BB} / 2. In the left, the multi-tone signal lies within the bandwidth of the interpolation filter with maximum flatness. In the right, although all the carriers are preserved at an slightly lower final sampling rate, the highest frequency tones show the effects of the resampling filter's roll off. The roll off of the resampling filter can be controlled by the number of taps so it can be more effective at the expense of calculation time.

The purpose of the resampling filter is to remove the images in the resampled waveform. Ideally, this filter should be a perfect "brick-wall" filter. Any real filter will approximate the ideal one and it will limit the available BW for the input waveform. Typically, the resampling filter is designed to provide a high enough useful bandwidth for the input waveform while making sure that the attenuation of the images is high enough. The useful bandwidth and the image attenuation depends on the number of taps of the resampling filter and calculation time for the resampled waveform is proportional to the number of taps. Resampling filters must be designed in such a way that the roll off is as gentle as possible while keeping the flatness over a sufficient portion of the first Nyquist Zone. As waveforms to resample must be bandwidth limited to a given fraction of the Nyquist bandwidth, the resampling filter roll-off can go beyond the Nyquist frequency so the required attenuation at the stop band is reached before any potential image shows up. This helps to increase the useful portion of the first Nyquist Zone, reduce the number of taps of the filter, or a combination of both effects.

3 Related Documentation

- Proteus Programming Manual
- Proteus Module User Manual
- Wave Design Studio User Manual
- Direct Generation/Acquisition of Microwave Signals, Tabor White Paper
- Effective Number of Bits for Arbitrary Waveform Generators, Tabor Application Note
- Digital Frequency Synthesis Demystified, Bar-Giora Goldberg

4 Appendix 1 – MATLAB Programming Example

```
% Baseband DUC example
% This is an example of how to generate signals and RF modulated signals
% simultaneously with the Proteus AWT. A complex modulated  RF signal is
% generated by one channel and the corresponding envelope signal is
% generated by another channel.

clear;
close all;
clear variables;
clear global;
clc;

% Define IP Address for Target Proteus device descriptor
% VISA "Socket-Based" TCP-IP Device. Socket# = 5025
ipAddr = '127.0.0.1'; %'127.0.0.1'= Local Host; % your IP here
pxiSlot = 0;

% Instrument setup
cType               = "LAN";  %"LAN" = VISA or "DLL" = PXI

if cType == "LAN"
    connPar         = ipAddr;
else
    connPar         = pxiSlot; % Your slot # here, o for manual selection
end

paranoia_level = 0; % 0, 1 or 2
% Open Session and load libraries
[inst, admin, model, slotNumber] = ConnecToProteus(cType, connPar,
paranoia_level);

% Report model
fprintf('Connected to: %s, slot: %d\n', model(1), slotNumber(1));

% Reset AWG
inst.SendScpi('*CLS;*RST');

% Get options using the standard IEEE-488.2 Command
optstr = getOptions(inst);

% AWG Settings
duc_iq_mode                             = 1; % 0 = HALF, 1 = ONE, 2 = TWO,
3 = NCO
sample_rate_dac                         = 9E9;
rf_channel                              = 1;
rf_segment                              = 1;
baseband_channel                        = 3;
baseband_segment                        = 3;
```

```
% Type of signal for test
% 1 = 802.11ax, 2 = 802.11ad, 3 = Multi-Tone, 4 = QAM
signal_type                             = 1;
carrier_freq                            = 2.412E9;
carrier_freq_2                          = 2.0E9; % For IQ Mode 2
baseband_mode                           = 2; % 1 = envelope, 2 = clock
(QAM)
% Envelope Tracking Settings
minimum_pwr                             = -20.0; % dB vs. peak power
smoothing_factor                        = 1000;

% Clock processing only makes sense for QAM
if signal_type ~= 4 && baseband_mode == 2
    baseband_mode = 1;
end

fprintf(1, 'BASEBAND WAVEFORM CALCULATION\n');

% Baseband waveform parameter definition
switch signal_type
    case 1
        interpolation_factor            = 8;
        actual_granularity              = 16;
        oversampling                    = 2;
        smoothing_factor                = 0.001;
        if baseband_mode == 2
            baseband_mode = 1;
        end

        [wfm_in, sample_rate_bb_in]     = Get_Wlan_ax(oversampling);
        wfm_in_2                        = wfm_in;
    case 2
        interpolation_factor            = 4;
        actual_granularity              = 32;
        smoothing_factor                = 0.005;
        if baseband_mode == 2
            baseband_mode = 1;
        end

        [wfm_in, sample_rate_bb_in]     = Get_Wlan_ad;
        wfm_in_2                        = wfm_in;
    case 3
        interpolation_factor            = 8;
        actual_granularity              = 16;
        num_of_tones                    = 40;
        offset_tone                     = 15;
        spacing                         = 1E6;
        oversampling                    = 1.1;
        smoothing_factor                = 1000; %0.05


        [wfm_in, sample_rate_bb_in]     = Get_Multi_Tone(   num_of_tones,
...
```

```
offset_tone,...
                                                              spacing, ...
                                                              oversampling);
        wfm_in_2                        = wfm_in;
    case 4
        interpolation_factor            = 8;
        actual_granularity              = 32;
        % modType           Modulation
        % 1                 QPSK
        % 2                 QAM16
        % 3                 QAM32
        % 4                 QAM64
        % 5                 QAM128
        % 6                 QAM256
        % 7                 QAM512
        % 8                 QAM1024
        modulation_type                 = 1; % QPSK
        num_of_symbols                  = 2^11;
        symbol_rate                     = 100E6; %50E6
        filter_type                     = 'sqrt'; % 'normal' or 'sqrt'
        roll_off                        = 0.15;
        oversampling                    = 6;
        smoothing_factor                = 0.001;

        [wfm_in, sample_rate_bb_in]     = Get_Qam(  modulation_type, ...
                                                    num_of_symbols, ...
                                                    symbol_rate, ...
                                                    filter_type,...
                                                    roll_off, ...
                                                    oversampling);
        % Second baseband waveform for IQ Mode 2. It must be consistent in
        % sampling rate and time window with first waveform
        modulation_type_2               = 2; % 16QAM
        num_of_symbols_2                = 2^12; % Twice the symbols
        symbol_rate_2                   = 100E6; %Twice the baud rate
        filter_type_2                   = 'sqrt'; % 'normal' or 'sqrt'
        roll_off_2                      = 0.25;
        oversampling_2                  = 3; % Half the oversampling
        smoothing_factor                = 0.001;

        [wfm_in_2, sample_rate_bb_in_2] = Get_Qam(  modulation_type_2, ...
                                                    num_of_symbols_2, ...
                                                    symbol_rate_2, ...
                                                    filter_type_2,...
                                                    roll_off_2, ...
                                                    oversampling_2);
        % For QAM and clock baseband signal, clock waveform must be
calculated
        if baseband_mode == 2
            baseband_wfm = Get_Qam_Clock(   num_of_symbols, ...
                                            roll_off, ...
                                            oversampling,...
                                            4);
```

```
        end

end
% Resampling must be carrier out for the DUC baseband sampling rate
sample_rate_bb_out = sample_rate_dac / interpolation_factor;
wfm_length_in = length(wfm_in);

%Calculation of lenght of the interpolated waveform
wfm_length_out = floor(wfm_length_in * sample_rate_bb_out /...
    (sample_rate_bb_in * actual_granularity)) * actual_granularity;

fprintf(1, 'BASEBAND WAVEFORM RESAMPLING\n');

%%%%%%%%%%%%%%%%%%%%%       RESAMPLING      %%%%%%%%%%%%%%%%%%%%%%%%%
wfm_out = myResampling(wfm_in, wfm_length_out, true, 60);
wfm_out_2 = myResampling(wfm_in_2, wfm_length_out, true, 60);

if signal_type == 4 && baseband_mode == 2
    % Clock waveform resampling
    baseband_wfm = myResampling(baseband_wfm, wfm_length_out, true, 60);
else
    % Get envelope tracking waveform from RF waveform
    [baseband_wfm, ref_envelope] = Get_Envelope(    wfm_out, ...
                                                    smoothing_factor, ...
                                                    minimum_pwr);
end

% Sample rate must be corected to compensate for the timing error
% introduced by the granularity requirements
actual_dac_sample_rate = wfm_length_out * interpolation_factor *...
    sample_rate_bb_in / wfm_length_in;

% Graph calculated waveforms in a proper way
fprintf(1, 'BASEBAND WAVEFORM GRAPHS\n');
if baseband_mode == 1
    % Show RF waveform in graph #1
    % And raw envelope and smoothed envelope in Graph #2
    DrawEnvelope(   wfm_out, ...
                    baseband_wfm, ...
                    ref_envelope, ...
                    sample_rate_bb_out);
else
    % Show unfiltered IQ and eye diagram in the top
    % and filtered IQ and eye diagram in the bottom
    DrawEyeDiagram( 3,...
                    1000, ...
                    actual_dac_sample_rate / interpolation_factor, ...
                    symbol_rate, ...
                    roll_off, ...
                    wfm_out, ...
                    baseband_wfm);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%% DOWNLOAD RF WAVEFORM
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
fprintf(1, 'RF WAVEFORM DOWNLOAD AND ACTIVATION\n');
% All previous waveforms will be deleted from waveform memory
inst.SendScpi(':TRAC:DEL:ALL');
% Format and download RF Signal
switch duc_iq_mode
    case 0
        result = SendIqmHalfWfm(inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                rf_channel,...
                                rf_segment,...
                                carrier_freq,...
                                0.0,...
                                true,...
                                wfm_out,...
                                16);
    case 1
        result = SendIqmOneWfm( inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                rf_channel,...
                                rf_segment,...
                                carrier_freq,...
                                0.0,...
                                true,...
                                wfm_out,...
                                16);
        result = SendIqmOneWfm( inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                rf_channel + 1,...
                                rf_segment + 1,...
                                carrier_freq,...
                                -90.0,...
                                true,...
                                wfm_out,...
                                16);
    case 2
        result = SendIqmTwoWfm( inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                rf_channel,...
                                rf_segment,...
                                carrier_freq,...
                                carrier_freq_2,...
                                0.0,...
                                0.0,...
                                true,...
                                wfm_out,...
                                wfm_out_2,...
                                16);
```

```
    case 3
       SetNco(  inst,...
                sample_rate_dac,...
                rf_channel,...
                carrier_freq,...
                0.0,...
                true);
%         for fr = 1E6:1E6:4500E6
%             inst.SendScpi(sprintf(':NCO:CFR1 %f', fr));
%         end


end

%%%%%%%%%%%%%%%%%%%%%%%%% DOWNLOAD BB WAVEFORM
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Format and download Baseband (Envelope or Clock) Signal
fprintf(1, 'BASEBAND WAVEFORM DOWNLOAD AND ACTIVATION\n');
switch duc_iq_mode
    case 0
        result = SendIqmHalfWfm(inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                baseband_channel,...
                                baseband_segment,...
                                0.0,...
                                0.0,...
                                true,...
                                baseband_wfm,...
                                16);

    case 1
        result = SendIqmOneWfm( inst,...
                                actual_dac_sample_rate,...
                                interpolation_factor,...
                                baseband_channel,...
                                baseband_segment,...
                                0.0,...
                                0.0,...
                                true,...
                                baseband_wfm,...
                                16);


end

% It is recommended to disconnect from instrument at the end
if cType == "LAN"
    inst.Disconnect();
else
    admin.CloseInstrument(inst.InstrId);
    admin.Close();
end
```

```
function result = SendIqmOneWfm(    inst,...
                                    samplingRate,...
                                    interpol,...
                                    channel,...
                                    segment,...
                                    cfr,...
                                    phase,...
                                    apply6db,...
                                    myWfm,...
                                    dacRes)

    % format Wfm and normalize waveform
    %myWfm = MyProteusInterpolation(myWfm, interpol, true);
    myWfm = NormalIq(myWfm);
    myWfm = Interleave(real(myWfm), imag(myWfm));
    myWfm = myQuantization(myWfm, dacRes, 1);

    % Select Channel
    inst.SendScpi(sprintf(':INST:CHAN %d', channel));

    inst.SendScpi([':FREQ:RAST ' num2str(2.5E9)]);
    % Interpolation factor for I/Q waveforms
    switch interpol
        case 2
            inst.SendScpi(':SOUR:INT X2');

        case 4
            inst.SendScpi(':SOUR:INT X4');

        case 8
            inst.SendScpi(':SOUR:INT X8');
    end

    % DAC Mode set to 'DUC' and IQ Modulation mode set to 'ONE'
    inst.SendScpi(':MODE DUC');
    inst.SendScpi(':IQM ONE');


    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);

    fprintf(1, sprintf('DOWNLOADING WAVEFORM: %d samples\n',
length(myWfm)));
    result = SendWfmToProteus(  inst,...
                                samplingRate,...
                                channel,...
                                segment,...
                                myWfm,...
                                dacRes,...
                                false);

    fprintf(1, 'WAVEFORM DOWNLOADED!\n');
    clear myWfm;

    % Select segment for generation
```

```
    fprintf(1, 'SETTING AWG OUTPUT\n');
    inst.SendScpi(sprintf(':FUNC:MODE:SEGM %d', segment));
    % Output volatge set to MAX
    inst.SendScpi(':SOUR:VOLT MAX');

    % NCO set-up
    % 6dB IQ Modulation gain applied
    if apply6db
        inst.SendScpi(':NCO:SIXD1 ON');
    else
        inst.SendScpi(':NCO:SIXD1 OFF');
    end
    % NCO frequency and phase setting
    inst.SendScpi(sprintf(':NCO:CFR1 %d', cfr));
    inst.SendScpi(sprintf(':NCO:PHAS1 %d', phase));

    % Activate outpurt and start generation
    inst.SendScpi(':OUTP ON');

    fprintf(1, 'SETTING SAMPLING CLOCK\n');
    % Set sampling rate for AWG as defined in the preamble.
    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
end

function result = SendIqmHalfWfm(   inst,...
                                    samplingRate,...
                                    interpol,...
                                    channel,...
                                    segment,...
                                    cfr,...
                                    phase,...
                                    apply6db,...
                                    myWfm,...
                                    dacRes)

    myWfm = NormalIq(myWfm);
    myWfmI = real(myWfm);
    myWfmI = myQuantization(myWfmI, dacRes, 1);
    myWfmQ = imag(myWfm);
    myWfmQ = myQuantization(myWfmQ, dacRes, 1);

    % Channel I is 2N - 1 and Channel Q is 2N
    % If channel is even, then base channle number is corrected
    if mod(channel, 2) == 0
        channel = channel - 1;
    end

    % Set temporary sampling rate for AWG.
    inst.SendScpi([':SOUR:FREQ:RAST ' num2str(2.5E9)]);

    res = inst.SendScpi('*OPC?');

    % The Half mode requires setting two channels
    inst.SendScpi(sprintf(':INST:CHAN %d', channel));
```

```
inst.SendScpi(':MODE DUC');
inst.SendScpi(':IQM HALF');

% Interpolation factor for I/Q waveforms
switch interpol
    case 2
        inst.SendScpi(':INT X2');

    case 4
        inst.SendScpi(':SOUR:INT X4');

    case 8
        inst.SendScpi(':SOUR:INT X8');
end

inst.SendScpi(sprintf(':INST:CHAN %d', channel + 1));

inst.SendScpi(':SOUR:MODE DUC');
inst.SendScpi(':SOUR:IQM HALF');

% Interpolation factor for I/Q waveforms
switch interpol
    case 2
        inst.SendScpi(':SOUR:INT X2');

    case 4
        inst.SendScpi(':SOUR:INT X4');

    case 8
        inst.SendScpi(':SOUR:INT X8');
end

inst.SendScpi([':SOUR:FREQ:RAST ' num2str(samplingRate)]);
% DAC Mode set to 'DUC' and IQ Modulation mode set to 'ONE';

% Waveform Downloading
% *******************
fprintf(1, 'DOWNLOADING WAVEFORM I\n');
result = SendWfmToProteus(  inst,...
                            samplingRate,...
                            channel,...
                            segment,...
                            myWfmI,...
                            dacRes,...
                            false);

fprintf(1, 'DOWNLOADING WAVEFORM Q\n');
result = SendWfmToProteus(  inst,...
                            samplingRate,...
                            channel + 1,...
                            segment + 1,...
                            myWfmQ,...
                            dacRes,...
```

```
                                    false);

    fprintf(1, 'WAVEFORMS DOWNLOADED!\n');
    clear myWfm;

    % Select segment for generation
    fprintf(1, 'SETTING AWG OUTPUT\n');
    % Q Channel
    inst.SendScpi(sprintf(':INST:CHAN %d', channel + 1));
    inst.SendScpi(sprintf(':FUNC:MODE:SEGM %d', segment + 1));
    % NCO frequency and phase setting
    inst.SendScpi(sprintf(':SOUR:NCO:CFR1 %d', cfr));
    inst.SendScpi(sprintf(':SOUR:NCO:PHAS1 %d', phase));
    if apply6db
        inst.SendScpi(':SOUR:NCO:SIXD1 ON');
    else
        inst.SendScpi(':SOUR:NCO:SIXD1 OFF');
    end

    % Output volatge set to MAX
    inst.SendScpi(':SOUR:VOLT 0.5');
    % Activate outpurt and start generation
    inst.SendScpi(':OUTP ON');

    % I Channel is set up in the end as this is the physical active output
    % I Channel
    inst.SendScpi(sprintf(':INST:CHAN %d', channel));
    inst.SendScpi(sprintf(':FUNC:MODE:SEGM %d', segment));
    % NCO frequency and phase setting
    inst.SendScpi(sprintf(':SOUR:NCO:CFR1 %d', cfr));
    inst.SendScpi(sprintf(':SOUR:NCO:PHAS1 %d', phase));
    if apply6db
        inst.SendScpi(':SOUR:NCO:SIXD1 ON');
    else
        inst.SendScpi(':SOUR:NCO:SIXD1 OFF');
    end

    % Output volatge set to MAX
    inst.SendScpi(':SOUR:VOLT 0.5');
    % Activate outpurt and start generation
    inst.SendScpi(':OUTP ON');

    fprintf(1, 'SETTING SAMPLING CLOCK\n');
    % Set sampling rate for AWG as defined in the preamble.
    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
end

function result = SendIqmTwoWfm(    inst,...
                                    samplingRate,...
                                    interpol,...
                                    channel,...
                                    segment,...
                                    cfr1,...
                                    cfr2,...
```

```
                                    phase1,...
                                    phase2,...
                                    apply6db,...
                                    myWfm1,...
                                    myWfm2,...
                                    dacRes)

    [myWfm1,  myWfm2] = NormalIq2(myWfm1, myWfm2);

    myWfm = formatWfm2(myWfm1, myWfm2);

    % Select Channel
    inst.SendScpi(sprintf(':INST:CHAN %d', channel));


    inst.SendScpi([':FREQ:RAST ' num2str(2.5E9)]);
    % Interpolation factor for I/Q waveforms
    switch interpol
        case 2
            inst.SendScpi(':SOUR:INT X2');

        case 4
            inst.SendScpi(':SOUR:INT X4');

        case 8
            inst.SendScpi(':SOUR:INT X8');
    end

    % DAC Mode set to 'DUC' and IQ Modulation mode set to 'ONE'
    % DAC Mode set to 'DUC' and IQ Modulation mode set to 'TWO'
    inst.SendScpi(':MODE DUC');
    inst.SendScpi(':IQM TWO');


    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);

    fprintf(1, sprintf('DOWNLOADING WAVEFORM: %d samples\n',
length(myWfm)));
    result = SendWfmToProteus(  inst,...
                                samplingRate,...
                                channel,...
                                segment,...
                                myWfm,...
                                dacRes,...
                                false);

    fprintf(1, 'WAVEFORM DOWNLOADED!\n');
    clear myWfm;

    % Select segment for generation
    fprintf(1, 'SETTING AWG OUTPUT\n');
    inst.SendScpi(sprintf(':FUNC:MODE:SEGM %d', segment));
    % Output volatge set to MAX
    inst.SendScpi(':SOUR:VOLT 0.5');
```

```
    % NCO set-up
    % 6dB IQ Modulation gain applied
    if apply6db
        inst.SendScpi(':NCO:SIXD1 ON');
        inst.SendScpi(':NCO:SIXD2 ON');
    else
        inst.SendScpi(':NCO:SIXD1 OFF');
        inst.SendScpi(':NCO:SIXD2 OFF');
    end
    % NCO frequency and phase setting
    inst.SendScpi(sprintf(':NCO:CFR1 %d', cfr1));
    inst.SendScpi(sprintf(':NCO:CFR2 %d', cfr2));
    inst.SendScpi(sprintf(':NCO:PHAS1 %d', phase1));
    inst.SendScpi(sprintf(':NCO:PHAS2 %d', phase2));

    % Activate outpurt and start generation
    inst.SendScpi(':OUTP ON');

    fprintf(1, 'SETTING SAMPLING CLOCK\n');
    % Set sampling rate for AWG as defined in the preamble.
    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
end

function SetNco(    inst,...
                    samplingRate,...
                    channel,...
                    cfr,...
                    phase,...
                    apply6db)

    % Select Channel
    inst.SendScpi(sprintf(':INST:CHAN %d', channel));
    fprintf(1, 'SETTING SAMPLING CLOCK\n');
    inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
    % DAC Mode set to 'NCO'
    inst.SendScpi(':MODE NCO');
    % 'NCO' Settings
    inst.SendScpi(sprintf(':NCO:CFR1 %d', cfr));
    inst.SendScpi(sprintf(':NCO:PHAS1 %d', phase));
    if apply6db
        inst.SendScpi(':NCO:SIXD1 ON');
    else
        inst.SendScpi(':NCO:SIXD1 OFF');
    end

    % Output volatge set to MAX
    inst.SendScpi(':SOUR:VOLT 0.5');
    % Activate outpurt and start generation
    inst.SendScpi(':OUTP ON');

    %fprintf(1, 'SETTING SAMPLING CLOCK\n');
    % Set sampling rate for AWG as defined in the preamble.
    %inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
```

```
end

function result = SendWfmToProteus( inst,...
                                    samplingRate,...
                                    channel,...
                                    segment,...
                                    myWfm,...
                                    dacRes,...
                                    initialize)

    if dacRes == 16
            inst.SendScpi(':TRAC:FORM U16');
    else
            inst.SendScpi(':TRAC:FORM U8');
    end

    %Select Channel
    if initialize
        inst.SendScpi(':TRAC:DEL:ALL');
        inst.SendScpi([':FREQ:RAST ' num2str(samplingRate)]);
    end

    inst.SendScpi(sprintf(':INST:CHAN %d', channel));
    inst.SendScpi(sprintf(':TRAC:DEF %d, %d', segment, length(myWfm)));
    % select segmen as the the programmable segment
    inst.SendScpi(sprintf(':TRAC:SEL %d', segment));

    % format Wfm
%     myWfm = myQuantization(myWfm, dacRes, 1);

    % Download the binary data to segment
    prefix = ':TRAC:DATA 0,';

    if (dacRes==16)
        myWfm = uint16(myWfm);
        myWfm = typecast(myWfm, 'uint8');
    else
        myWfm = uint8(myWfm);
    end
    tic;
    %res = inst.WriteBinaryData(':TRAC:DATA ', myWfm);
    res = inst.WriteBinaryData(prefix, myWfm);

    assert(res.ErrCode == 0);

%     if dacRes == 16
%         inst.SendBinaryData(prefix, myWfm, 'uint16');
%     else
%         inst.SendBinaryData(prefix, myWfm, 'uint8');
%     end

    if initialize
        inst.SendScpi(sprintf(':SOUR:FUNC:MODE:SEGM %d', segment))
        % Output voltage set to MAX
```

```
        inst.SendScpi(':SOUR:VOLT 0.5');
        % Activate outpurt and start generation
        inst.SendScpi(':OUTP ON');
    end

    result = length(myWfm);
end

function resampling_filter = GetResamplingFilter(
num_of_convolution_samples, ...
                                                    resolution_of_filter,
...
                                                    bw_fraction)
    % Creation of sinc lookup table
    % The NumOfConvolutionSamples paramters controls the quality of the
    % resampling filter in terms of roll-off and attenuation at the stop
    % band. The more, the better quality, the longer calculation time.
    % ResFilter sets the number of values per sample time to be included
in
    % the look-up table. The more, the better quality, the longer
    % calculation time.
    % bwFrac reduces de BW of the filter to avoid aliasing problems caused
    % by the roll-off of the resampling filter.
    % A resampling filter object is created with the lookup table for it
    % (just one side as it is symmetrical) and all the associated
    % parameters.
    resampling_filter.num_of_samples = num_of_convolution_samples;
    resampling_filter.resolution = resolution_of_filter;
    resampling_filter.bw_fraction = bw_fraction;
    sinc_length = floor(num_of_convolution_samples * resolution_of_filter
/ bw_fraction);
    resampling_filter.filter = 0:(sinc_length);
    resampling_filter.filter = resampling_filter.filter /
resolution_of_filter;
    resampling_filter.filter = resampling_filter.filter * bw_fraction;
    % Basic filter shape is ideal low pass filter (sinc)
    resampling_filter.filter = sinc(resampling_filter.filter);
    % Flattop window is applied to improve flatness and stop band
rejection
    windowed_filter = flattopwin(2 * sinc_length);
    windowed_filter = windowed_filter(sinc_length:end);
    resampling_filter.filter = resampling_filter.filter .*
windowed_filter';
end

function output_wfm = myResampling (    input_wfm, ...
                                        output_wfm_length, ...
                                        is_circular, ...
                                        quality, ...
                                        resampling_filter)
% This funtion resamples the input waveform (inWfm) to generate a new
% waveform with a new length (outWl). New length can be longer
(upsampling)
% or shorter (downsampling) than the original one. The new waveform can be
```

```
% selfconsistent for loop generation (isCirc == true) or not for singe
shot
% generation.

    input_wfm_length = length(input_wfm);
    % Sampling rate ratio (>1.0, upsampling)
    sampling_ratio = double(output_wfm_length) / double(input_wfm_length);
    % If resampling filter exists it is not calculated so time is saved
    % when calling the resampling function more than once
    if ~exist('resampling_filter', 'var') ||  isempty(resampling_filter)
        % Default parameters for resampling filter
        filter_resolution = 50000; %50000
        bw_fraction = 1.0; %0.98;
        if sampling_ratio < 1.0
            bw_fraction = 0.98;
        end
        resampling_filter = GetResamplingFilter(    quality, ...
                                                    filter_resolution, ...
                                                    bw_fraction);
    end
    % The parameters of the resampling filter are part of the associated
    % object
    convolution_length = resampling_filter.num_of_samples;
    filter_resolution = resampling_filter.resolution;
    resampling_filter_length = length(resampling_filter.filter);
    bw_fraction = resampling_filter.bw_fraction;
    % For undersampling filter, the amplitude of the resampling filter
must
    % be corrected by the relative BW
    if sampling_ratio < 1.0
        % The distance for samples in the input (measured in samples of
the
        % output) must be corrected for undersampling as well in order to
        % preserve SFDR
        convolution_length = floor(convolution_length / (sampling_ratio *
bw_fraction));
        resampling_filter.filter = resampling_filter.filter *
(sampling_ratio * bw_fraction);
    else
        convolution_length = floor(resampling_filter.num_of_samples /
bw_fraction);
    end

    % Output waveform is initialized to "all zeros"
    output_wfm = zeros(1, output_wfm_length);
    % Convolution loop for each output sample
    if sampling_ratio >= 1.0
        mult_factor1 = bw_fraction * filter_resolution;
    else
        mult_factor1 = bw_fraction * filter_resolution * sampling_ratio;
    end

    for i = 0:(output_wfm_length - 1)
        % Index for the central sample to process in the input wfm
```

```
        central_sample = i / sampling_ratio;
        central_sample_int = round(central_sample);
        % Contribution for all the participating samples form the input is
        % accumulated on the current output sample
        for j = (central_sample_int - convolution_length):...
                (central_sample_int + convolution_length)
            % Actual fractional distance to the input sample
            time_distance = abs(central_sample - j);

            % Distance is converted to a relative integer index to the
            % resampling filter (lookup table)
            time_distance = round(mult_factor1 * time_distance);

            % If convolution is circular the initial samples are used at
            % the end and the end samples are used at the beginning.
            input_wfm_index = j;
            if is_circular
                input_wfm_index = mod(input_wfm_index, input_wfm_length);
            end
            % If the pointer to the resampling filter is within teh limits
            % of the lookup table, the contribution of the input sample is
            % added to the current output sample
            if time_distance < resampling_filter_length && ...
                    input_wfm_index >=0 && ...
                    input_wfm_index < input_wfm_length
                output_wfm(i + 1) = output_wfm(i + 1) +...
                    input_wfm(input_wfm_index + 1) * ...
                    resampling_filter.filter(time_distance + 1);
            end
        end
    end
end

function output_wfm = LimitBW ( input_wfm, ...
                                bw_fraction)
    num_of_peak_samples = round(1.0 / bw_fraction);
    output_wfm = input_wfm;
    for k = 0:(length(input_wfm) - 1)
        ref_sample = k + 1;
        for j = (k - num_of_peak_samples):(k + num_of_peak_samples)
            current_sample = int32(mod(j, length(input_wfm)) + 1);
            if input_wfm(current_sample) > output_wfm(ref_sample)
                output_wfm(ref_sample) = input_wfm(current_sample);
            end
        end
    end
end

function [waveform, Fs] = Get_Wlan_ad()
% Generated by MATLAB(R) 9.14 (R2023a) and WLAN Toolbox 3.6 (R2023a).
% Generated on: 19-Apr-2023 18:52:47

    %% Generating 802.11ad waveform
    % 802.11ad configuration
```

```
    dmgCfg = wlanDMGConfig('MCS', '16', ...
        'TrainingLength', 0, ...
        'TonePairingType', 'Static', ...
        'PSDULength', 1000, ...
        'AggregatedMPDU', false, ...
        'LastRSSI', 0, ...
        'Turnaround', false);

    num_of_packets = 1;
    idle_time = 2E-6;
    % input bit source:
    in = randi([0, 1], 1000, 1);

    % Generation
    waveform = wlanWaveformGenerator(in, dmgCfg, ...
        'NumPackets', num_of_packets, ...
        'IdleTime', idle_time, ...
        'WindowTransitionTime', 6.0606e-09, ...
        'ScramblerInitialization', 2);

    Fs = wlanSampleRate(dmgCfg); % Specify the sample rate of the waveform
in Hz
end

function [waveform, Fs] = Get_Wlan_ax(oversampling)
    % 802.11ax configuration
    heSUCfg = wlanHESUConfig('ChannelBandwidth', 'CBW160', ...
        'NumTransmitAntennas', 1, ...
        'NumSpaceTimeStreams', 1, ...
        'SpatialMapping', 'Direct', ...
        'PreHESpatialMapping', false, ...
        'MCS', 5, ...
        'DCM', false, ...
        'ChannelCoding', 'LDPC', ...
        'APEPLength', 100, ...
        'GuardInterval', 3.2, ...
        'HELTFType', 4, ...
        'UplinkIndication', false, ...
        'BSSColor', 0, ...
        'SpatialReuse', 0, ...
        'TXOPDuration', 127, ...
        'HighDoppler', false, ...
        'NominalPacketPadding', 0);

    % input bit source:
    in = randi([0, 1], 10000, 1);

    num_of_packets = 1;
    idle_time = 20E-6;

    % Generation
    waveform = wlanWaveformGenerator(in, heSUCfg, ...
        'NumPackets', num_of_packets, ...
        'IdleTime', idle_time, ...
```

```
        'OversamplingFactor', oversampling, ...
        'ScramblerInitialization', 93, ...
        'WindowTransitionTime', 1e-07);

    Fs = oversampling * wlanSampleRate(heSUCfg, 'OversamplingFactor', 1);
end


function [waveform, Fs] = Get_Multi_Tone(   num_of_tones, ...
                                            offset_tone,...
                                            spacing, ...
                                            oversampling)
    % Compute maximum frequency component in the signal
    max_freq = (num_of_tones - 1) * spacing / 2.0;
    max_freq = max_freq + spacing * offset_tone;
    % Sample rate for calcualtion will be twice the maximum freq x
    % oversampling factor
    Fs = oversampling * 2.0 * max_freq;
    % Tone frequency calculation
    tone_freq = 0:(num_of_tones - 1);
    tone_freq = tone_freq - (num_of_tones - 1.0) / 2.0;
    tone_freq = spacing * tone_freq;
    tone_freq = tone_freq + spacing * offset_tone;

    % Time window will be the minimum one: 1 / spacing
    % It must be double when the number of tones is even for symmetrical
    % spectrum around carrier frequency.
    if mod(num_of_tones,2) == 1
        time_window = 1.0 / spacing;
    else
        time_window = 2.0 / spacing;
    end

    % Waveform length must be an integer
    wfm_length = round(Fs * time_window);
    % Fs must be recalculated after rounding wavweform length
    Fs = wfm_length / time_window;
    % Time values for samples
    x_data = 0 :(wfm_length -1);
    x_data = x_data / Fs;
    % Phase distribution for PAPR reduction is selected. Newman = 2.
    tones_phase = PhaseDistribution(2, num_of_tones);
    % Waveform data is initialized to zero
    waveform = zeros(1, wfm_length);
    % The contribution of each tone is added to the waveform
    for k = 1 : num_of_tones
        waveform = waveform + ...
            exp(1i * (x_data * 2 * pi * tone_freq(k) + tones_phase(k)));
    end
end

function phase_table = PhaseDistribution(dist_type, number_of_tones)
    switch dist_type
        case 1
```

```
            % Random
            phase_table = 2.0 * pi .* (rand(1, number_of_tones) - 0.5);

        case 2
            % Newman (near-optimal for equal amplitude tones)
            phase_table = 1:number_of_tones;
            phase_table = wrapToPi(-(pi / number_of_tones) .* ...
                ( 1.0 - phase_table .* phase_table));

        case 3
            % Rudin (near optimal for equal amplitude tones when number of
            % tones = 2^N)
            num_of_steps = int16(round(log(number_of_tones) / log(2)));

            if 2^num_of_steps < number_of_tones
                num_of_steps = num_of_steps + 1;
            end

            num_of_steps = num_of_steps - 1;
            phase_table(1:2) = 1;
            % Rudin sequence construction
            for n=1:num_of_steps
                m = int16(length(phase_table) / 2);
                phase_table = [phase_table, phase_table(1 : m),...
                    -phase_table(m + 1 : 2 * m)];
            end
            % Conversion to radians
            phase_table = -0.5 * pi .* (phase_table(1 : number_of_tones) -
1);
    end
end

function [symbol] = getIqMap(data, bPerS)

    if bPerS == 5 % QAM32 mapping
        lev = 6;
        data = data + 1;
        data(data > 4) = data(data > 4) + 1;
        data(data > 29) = data(data > 29) + 1;

    elseif bPerS == 7 % QAM128 mapping
        lev = 12;
        data = data + 2;
        data(data > 9) = data(data > 9) + 4;
        data(data > 21) = data(data > 21) + 2;
        data(data > 119) = data(data > 119) + 2;
        data(data > 129) = data(data > 129) + 4;

     elseif bPerS == 9 % QAM512 mapping
        lev = 24;
        data = data + 4;
        data(data > 19) = data(data > 19) + 8;
        data(data > 43) = data(data > 43) + 8;
        data(data > 67) = data(data > 67) + 8;
```

```
        data(data > 91) = data(data > 91) + 4;
        data(data > 479) = data(data > 479) + 4;
        data(data > 499) = data(data > 499) + 8;
        data(data > 523) = data(data > 523) + 8;
        data(data > 547) = data(data > 547) + 8;
    else
        lev = 2 ^ (bPerS / 2); % QPSK, QAM16, QAM64, QAM256, QAM1024
    end

    symbI = floor(data / lev);
    symbQ = mod(data, lev);
    lev = lev / 2 - 0.5;
    symbI = (symbI - lev) / lev;
    symbQ = (symbQ - lev) / lev;
    symbol = symbI + 1i * symbQ;
end

function dataOut = getRnData(nOfS, bPerS)
    maxVal = 2 ^ bPerS;
    dataOut = maxVal * rand(1, nOfS);
    dataOut = floor(dataOut);
    dataOut(dataOut >= maxVal) = maxVal - 1;
end

function out_vector = ZeroPadding(in_vector, oversampling)
    out_vector = zeros(1, oversampling * length(in_vector));
    out_vector(1:oversampling:length(out_vector)) = in_vector;
end

function [waveform, Fs] = Get_Qam(  modulation_type, ...
                                    num_of_symbols, ...
                                    symbol_rate, ...
                                    filter_type,...
                                    roll_off, ...
                                    oversampling)

    % modType            Modulation
    % 1                  QPSK
    % 2                  QAM16
    % 3                  QAM32
    % 4                  QAM64
    % 5                  QAM128
    % 6                  QAM256
    % 7                  QAM512
    % 8                  QAM1024

    bits_per_symbol = [2, 4, 5, 6, 7, 8, 9, 10];
    bits_per_symbol = bits_per_symbol(modulation_type);
    oversampling = round(oversampling);

    % Create IQ for QPSK/QAM

    % Get symbols in the range 1..2^bps and Map to IQ as Complex Symbol
    data = getRnData(num_of_symbols, bits_per_symbol);
    [waveform] = getIqMap(data, bits_per_symbol);
```

```
    % Adapt I/Q sample rate to the Oversampling parameter
    waveform = ZeroPadding(waveform, oversampling);

    % Calculate baseband shaping filter
    % accuracy is the length of-1 the shaping filter
    accuracy = 512;
    %filter_type = 'sqrt'; % 'normal' or 'sqrt'
    baseband_filter = rcosdesign(   roll_off, ...
                                    accuracy, ...
                                    oversampling, ...
                                    filter_type);

    % Apply filter through circular convolution and calculate Fs
    waveform = cconv(waveform, baseband_filter, length(waveform));
    Fs = symbol_rate * oversampling;
end

function [envelope_wfm, ref_envelope] = Get_Envelope(wfm_out,
smoothing_factor, minimum_pwr)
% ENVELOPE CALCULATION
    % Envelope wfm made from the module of the IQ complex wfm
    envelope_wfm = abs(wfm_out);
    % LPF
    envelope_wfm = LimitBW( envelope_wfm, smoothing_factor);
    envelope_wfm = movmean(envelope_wfm, 10);
    %envelope_wfm = LimitBW( envelope_wfm, bw_factor);
    % Minimum level processing
    minimum_pwr = max(envelope_wfm) * 10^(minimum_pwr / 20.0);
    envelope_wfm(envelope_wfm < minimum_pwr) = minimum_pwr;
    % Normalization so 0 will be mapped to the lowest DAC value and max is
    % mapped to +1.0. wfm_out is always positive
    if max(envelope_wfm) > 0.0
        envelope_wfm = 2.0 * (envelope_wfm / max(envelope_wfm) - 0.5);
    else
        envelope_wfm = envelope_wfm + 1.0;
    end

    ref_envelope = abs(wfm_out);
    if max(ref_envelope) > 0.0
        ref_envelope = 2.0 * (ref_envelope / max(ref_envelope) - 0.5);
    else
        ref_envelope = ref_envelope + 1.0;
    end

end

function waveform = Get_Qam_Clock(  num_of_symbols, ...
                                    roll_off, ...
                                    oversampling, ...
                                    div_factor)

    oversampling = round(oversampling);
    div_factor = round(div_factor);
```

```
    % Create IQ for QPSK/QAM

    % Get symbols in the range 1..2^bps and Map to IQ as Complex Symbol
    waveform = zeros(1, num_of_symbols);
    for k = 0:(div_factor - 1)
        waveform((k + 1):(2 * div_factor):length(waveform)) = 1.0;
    end
    for k = div_factor:(2 * div_factor - 1)
        waveform((k + 1):(2 * div_factor):length(waveform)) = -1.0;
    end

    % Adapt I/Q sample rate to the Oversampling parameter
    waveform = ZeroPadding(waveform, oversampling);

    % Calculate baseband shaping filter
    % accuracy is the length of-1 the shaping filter
    accuracy = 512;
    filter_type = 'sqrt'; % 'normal' or 'sqrt'
    baseband_filter = rcosdesign(   roll_off, ...
                                    accuracy, ...
                                    oversampling, ...
                                    filter_type);

    % Apply filter through circular convolution and calculate Fs
    waveform = cconv(waveform, baseband_filter, length(waveform));
end

function DrawEnvelope(  wfm_out, ...
                        envelope_wfm, ...
                        ref_envelope, ...
                        sample_rate_bb_out)

    % Two plots
    tiledlayout(1,2);

    x0=100;
    y0=100;
    width=1000;
    height=800;
    set(gcf,'position',[x0,y0,width,height]);

    wfm_length_out = length(wfm_out);

    nexttile;
    x_data = 0 : (wfm_length_out - 1);
    x_data = x_data / sample_rate_bb_out;
    plot(x_data, real(wfm_out));
    hold;
    plot(x_data, imag(wfm_out));
    title(strcat('IQ Waveform:', num2str(wfm_length_out),' samples @',...
        num2str(sample_rate_bb_out / 1E6), 'MS/s'));
    xlabel('Seconds');
```

```
    nexttile;
    plot(x_data, ref_envelope);
    hold;
    plot(x_data, envelope_wfm);
    ylim([-1.0 1.1]);
    title(strcat('Envelope Waveform:', num2str(wfm_length_out),' samples
@',...
        num2str(sample_rate_bb_out / 1E6), 'MS/s'));
    xlabel('Seconds');
end

function DrawEyeDiagram(    eye_width, ...
                            max_symbol_shown, ...
                            sample_rate, ...
                            symbol_rate, ...
                            roll_off, ...
                            wfm_in, ...
                            clock_wfm)

    % For better graph accuracy, samples per symbol > = 100
    interpol_factor = ceil(sample_rate / symbol_rate);
    if interpol_factor < 100
        interpol_factor = ceil(100 / interpol_factor);
        new_wfm_length = interpol_factor * length(wfm_in);
        wfm_in = myResampling(wfm_in, new_wfm_length, true, 60);
        clock_wfm = myResampling(clock_wfm, new_wfm_length, true, 60);
        sample_rate = interpol_factor * sample_rate;
    end
    % Graph data definition
    size_window_in_samples = ceil(eye_width / symbol_rate * sample_rate);
    size_window_in_samples = ceil(size_window_in_samples / 2);
    symbol_shift = round(0.5 / symbol_rate * sample_rate);
    % Zero crossing for clock signal
    zero_crossings = zeros(1, max_symbol_shown);
    previous_state = clock_wfm(1);

    filter_type = 'sqrt'; % 'normal' or 'sqrt'
    baseband_filter = rcosdesign(roll_off, 60, ...
        round(sample_rate / symbol_rate) , filter_type);
    baseband_filter = baseband_filter / sum(baseband_filter);

    % Zero crossing processing
    counter = 1;

    for k = 2:length(clock_wfm)
        if clock_wfm(k) >= 0.0 && previous_state <= 0.0 ||...
            clock_wfm(k) <= 0.0 && previous_state >= 0.0
            zero_crossings(counter) = k - 1;
            previous_state = clock_wfm(k);
            if counter > max_symbol_shown
                break;
            else
                counter = counter + 1;
            end
```

```
        end
    end

    % Four plots
    tiledlayout(2,2);

    x0 = 100;
    y0 = 100;
    width = 1000;
    height = 800;
    set(gcf,'position',[x0,y0,width,height]);

    nexttile;
    plot(wfm_in);
    hold;
    const_diagram = wfm_in(zero_crossings(2:counter - 2) + symbol_shift);
    scatter(real(const_diagram), imag(const_diagram), 20, [1, 1, 0],
'filled');

    max_ampl = max([max(abs(real(wfm_in))), max(abs(imag(wfm_in)))]);

    xlim([-max_ampl max_ampl]);
    ylim([-max_ampl max_ampl]);
    title('Constellation Unfiltered');

    nexttile;

    base_x_data = -size_window_in_samples:1:size_window_in_samples;
    base_x_data = base_x_data / sample_rate;
    hold_flag = true;

    for k = 1:counter
        if (zero_crossings(k) - size_window_in_samples) >= 1 &&...
                (zero_crossings(k) + size_window_in_samples) <=
length(wfm_in)
            plot(   base_x_data, ...
                    real(wfm_in(zero_crossings(k) - size_window_in_samples
+ symbol_shift:...
                                zero_crossings(k) + size_window_in_samples
+ symbol_shift)));
            if hold_flag
                hold on;
                hold_flag = false;
            end
        end
    end

    title('Eye Diagram Unfiltered');
    xlabel('Symbol Period');

    nexttile;
    % Apply filter through circular convolution
    clock_wfm = cconv(clock_wfm, baseband_filter, length(clock_wfm));
    % Apply filter through circular convolution and calculate Fs
```

```
    wfm_in = cconv(wfm_in, baseband_filter, length(wfm_in));

    counter = 1;

    for k = 2:length(clock_wfm)
        if clock_wfm(k) >= 0.0 && previous_state < 0.0 ||...
                clock_wfm(k) < 0.0 && previous_state >= 0.0
            zero_crossings(counter) = k;
            previous_state = clock_wfm(k);
            if counter > max_symbol_shown
                break;
            else
                counter = counter + 1;
            end
        end
    end

    plot(wfm_in);
    const_diagram = wfm_in(zero_crossings(2:counter - 2) + symbol_shift);
    hold;
    scatter(real(const_diagram), imag(const_diagram), 20, [1, 1, 0],
'filled');

    max_ampl = max([max(abs(real(wfm_in))), max(abs(imag(wfm_in)))]);

    xlim([-max_ampl max_ampl]);
    ylim([-max_ampl max_ampl]);

    title('Constellation Filtered');

    nexttile;

    hold_flag = true;

    for k = 1:counter
        if (zero_crossings(k) - size_window_in_samples) >= 1 &&...
                (zero_crossings(k) + size_window_in_samples) <=
length(wfm_in)
            plot(base_x_data, real(wfm_in(zero_crossings(k) -
size_window_in_samples + symbol_shift:...
                zero_crossings(k) + size_window_in_samples +
symbol_shift)));
            if hold_flag
                hold on;
                hold_flag = false;
            end
        end
    end

    title('Eye Diagram Filtered');
    xlabel('Symbol Period');
end

function [  inst,...
```

```
            admin,...
            modelName,...
            sId] = ConnecToProteus( cType, ...
                                    connStr, ...
                                    paranoia_level)

% Connection to target Proteus
% cType specifies API. "LAN" for VISA, "DLL" for PXI
% connStr is the slot # as an integer(0 for manual selection) or IP adress
% as an string
% Paranoia Level add additional checks for each transfer. 0 = no checks.
% 1 = send OPC?, 2 = send SYST:ERROR?

% It returns
% inst: handler for the selected instrument
% admin: administrative handler
% modelName: string with model name for selected instrument (i.e. "P9484")
% sId: slot number for selected instrument

    pid = feature('getpid');
    fprintf(1,'\nProcess ID %d\n',pid);

    dll_path = 'C:\\Windows\\System32\\TEPAdmin.dll';
    admin = 0;
    sId = 0;
    if cType == "LAN"
        try
            connStr = strcat('TCPIP::',connStr,'::5025::SOCKET');
            inst = TEProteusInst(connStr, paranoia_level);

            res = inst.Connect();
            assert (res == true);
            modelName = identifyModel(inst);
        catch ME
            rethrow(ME)
        end
    else
        asm = NET.addAssembly(dll_path);

        import TaborElec.Proteus.CLI.*
        import TaborElec.Proteus.CLI.Admin.*
        import System.*

        admin = CProteusAdmin(@OnLoggerEvent);
        rc = admin.Open();
        assert(rc == 0);

        try
            slotIds = admin.GetSlotIds();
            numSlots = length(size(slotIds));
            assert(numSlots > 0);

            % If there are multiple slots, let the user select one ..
            sId = slotIds(1);
```

```
            if numSlots > 1
                fprintf('\n%d slots were found\n', numSlots);
                for n = 1:numSlots
                    sId = slotIds(n);
                    slotInfo = admin.GetSlotInfo(sId);
                    if ~slotInfo.IsSlotInUse
                        modelName = slotInfo.ModelName;
                        if slotInfo.IsDummySlot && connStr == 0
                            fprintf(' * Slot Number:%d Model %s [Dummy
Slot].\n', sId, modelName);
                        elseif connStr == 0
                            fprintf(' * Slot Number:%d Model %s.\n', sId,
modelName);
                        end
                    end
                end
                pause(0.1);
                if connStr == 0
                    choice = input('Enter SlotId ');
                    fprintf('\n');
                else
                    choice = connStr;
                end
                sId = uint32(choice);
                slotInfo = admin.GetSlotInfo(sId);
                modelName = slotInfo.ModelName;
                modelName = strtrim(netStrToStr(modelName));
            end

            % Connect to the selected instrument ..
            should_reset = true;
            inst = admin.OpenInstrument(sId, should_reset);
            instId = inst.InstrId;

        catch ME
            admin.Close();
            rethrow(ME)
        end
    end
end

function model = identifyModel(inst)
    idnStr = inst.SendScpi('*IDN?');
    idnStr = strtrim(netStrToStr(idnStr.RespStr));
    idnStr = split(idnStr, ',');

    if length(idnStr) > 1
        model = idnStr(2);
    else
        model ='';
    end
end

function options = getOptions(inst)
```

```
    optStr = inst.SendScpi('*OPT?');
    optStr = strtrim(netStrToStr(optStr.RespStr));
    options = split(optStr, ',');
end

function [str] = netStrToStr(netStr)
    try
        str = convertCharsToStrings(char(netStr));
    catch
        str = '';
    end
end

function retval = myQuantization (myArray, dacRes, minLevel)

    maxLevel = 2 ^ dacRes - 1;
    numOfLevels = maxLevel - minLevel + 1;

    retval = round((numOfLevels .* (myArray + 1) - 1) ./ 2);
    retval = retval + minLevel;

    retval(retval > maxLevel) = maxLevel;
    retval(retval < minLevel) = minLevel;

end

function outWfm = Interleave2(wfmI, wfmQ)

    wfmLength = length(wfmI);
    if length(wfmQ) < wfmLength
        wfmLength =  length(wfmQ);
    end

    %wfmLength = 2 * wfmLength;
    outWfm = uint8(zeros(1, 2 * wfmLength));

    outWfm(1:2:(2 * wfmLength - 1)) = wfmI;
    outWfm(2:2:(2 * wfmLength)) = wfmQ;
end

function outWfm = formatWfm2(inWfm1, inWfm2)
%formatWfm2 This function formats data for two I/Q streams to be dwnloaded
%to a single segment in Proteus to be generated in the IQM Mode 'TWO'
%   All waveforms must be properly normalized to the -1.0/+1.0 range.
%   All waveforms must have the same length

    % Formatting requires to go through the following steps:
    %   1) quantize samples to 16-bit unsigned integers
    %   2) swap the LSB and MSB as MSBs will be sent first for this mode
    %   3) convert the uint16 array to an uint8 array of twice the size
    % Final wfm is MSB, LSB, MSB, LSB,...
    inWfmI1 = typecast(swapbytes(uint16(myQuantization(real(inWfm1), 16,
1))),'uint8');
```

```
    inWfmQ1 = typecast(swapbytes(uint16(myQuantization(imag(inWfm1), 16,
1))),'uint8');
    inWfmI2 = typecast(swapbytes(uint16(myQuantization(real(inWfm2), 16,
1))),'uint8');
    inWfmQ2 = typecast(swapbytes(uint16(myQuantization(imag(inWfm2), 16,
2))),'uint8');
    % Sequence MSBI1, MSBQ1, MSBQ2, MSBI2, LSBI1, LSBQ1, LSBQ2, LSBI2
    % This is done in three interleaving steps
    outWfmI = Interleave2(inWfmI1, inWfmQ2);
    outWfm = Interleave2(inWfmQ1, inWfmI2);
    outWfm = Interleave2(outWfmI, outWfm);

    % Format as 16 bit integers as this is how waveforms are transferred
    outWfm = uint16(outWfm(1:2:length(outWfm))) + ...
            256 * uint16(outWfm(2:2:length(outWfm)));
end

function shifted_vector = ShiftVector(input_wfm, shifts)

    vector_l = length(input_wfm);
    shifts = shifts - 1;
    shifted_vector = input_wfm(mod((1:vector_l) + shifts, vector_l) + 1);
end

function zeroed_vector = InsertZeros(input_vector, isEven)

    if isEven
        zeroed_vector = zeros(1, 2 * length(input_vector));
    else
        zeroed_vector = zeros(1, 2 * length(input_vector) - 1);
    end

    zeroed_vector(1:2:length(zeroed_vector)) = input_vector;
end

function [interpol_filter, max_response] =
GetProteusInterpolFilter(interpolation_factor)
    basic_2x_filter_taps = [6, 0, -19, 0, 47, 0, -100, 0, 192, 0, -342,
0,...
    572, 0, -914, 0, 1409, 0, -2119, 0, 3152, 0, -4729, 0, 7420, 0,...
    -13334, 0, 41527, 65536, 41527, 0, -13334, 0, 7420, 0, -4729, 0,...
    3152, 0, -2119, 0, 1409, 0, -914, 0, 572, 0, -342, 0, 192, 0, -100,...
    0, 47, 0, -19, 0, 6];

    switch interpolation_factor

        case 2
            interpol_filter = basic_2x_filter_taps;

        case 4
            interpol_filter = InsertZeros(basic_2x_filter_taps, false);
            interpol_filter = conv(interpol_filter, basic_2x_filter_taps);

        case 8
```

```
            interpol_filter = InsertZeros(basic_2x_filter_taps, false);
            interpol_filter = conv(interpol_filter, basic_2x_filter_taps);
            interpol_filter = InsertZeros(interpol_filter, false);
            interpol_filter = conv(interpol_filter, basic_2x_filter_taps);
    end

    % Filter normalization for 0dB gain at 0Hz
    interpol_filter = interpol_filter/sum(interpol_filter);
    interpol_filter = interpolation_factor * interpol_filter;

    % Worst case maximum output abs(amplitude) for
    max_response = 0.0;

    for k = 0:(interpolation_factor - 1)
        current_max_response =
sum(abs(interpol_filter((k+1):interpolation_factor:length(interpol_filter)
)));
        if current_max_response > max_response
            max_response = current_max_response;
        end
    end
end

function output_wfm = MyProteusInterpolation(input_wfm, interpol_factor,
apply_norm)
      % Function used in traditional resampling
      % Expansion by zero-padding
      output_wfm = zeros(1, interpol_factor * length(input_wfm));
      output_wfm(1:interpol_factor:end) = input_wfm;
      % "Ideal" Interpolation filter
      [interpol_filter, max_response] =
GetProteusInterpolFilter(interpol_factor);
      shifts = floor(length(interpol_filter) / 2);

      %convolution
      output_wfm = cconv(output_wfm, interpol_filter, length(output_wfm));
      output_wfm = ShiftVector(output_wfm, shifts);
      if apply_norm
          output_wfm = input_wfm /  max(abs(output_wfm));
      end
end

function outWfm = NormalIq(wfm)
    maxPwr = max(abs(wfm));
    outWfm = wfm / maxPwr;
end

function [outWfm1,  outWfm2] = NormalIq2(wfm1, wfm2)
    maxPwr = max(abs(wfm1) + abs(wfm2));
    outWfm1 = wfm1 / maxPwr;
    outWfm2 = wfm2 / maxPwr;
end

function outWfm = Interleave(wfmI, wfmQ)
```

```
    wfmLength = length(wfmI);
    outWfm = zeros(1, 2 * wfmLength);

    outWfm(1:2:(2 * wfmLength - 1)) = wfmI;
    outWfm(2:2:(2 * wfmLength)) = wfmQ;
end
```