RF Signal Generation with Digital Up-Converters in AWGs

White Paper

Rev. 1.0

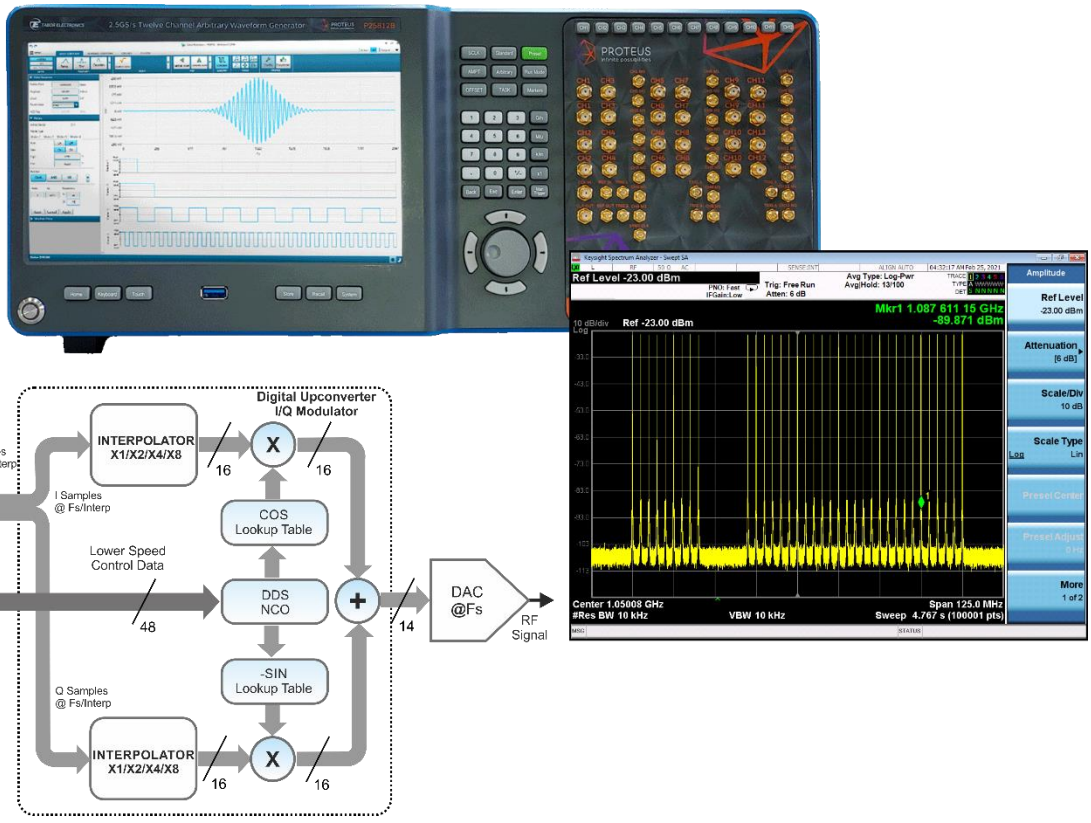


Table of Contents

1 RF Signal Generation Using AWGs

Arbitrary Waveform Generators (AWG) have always been incorporated in RF signal generation systems to generate complex modulations, analog or digital. Traditionally, AWGs generated real or complex (I/Q) baseband signals to feed modulators. In particular, quadrature (IQ) modulators combined with 2-channel AWGs can generate any analog or digital modulation, provided the modulation bandwidth of the modulator and the bandwidth/sampling rate of the AWG are sufficient to faithfully generate the desired signal (fig. 1.1a). IQ modulators are very sensitive to differential responses for the I and Q signal path, no matter if they come from the AWG or the modulator. Any imbalance, quadrature, I/Q skew, etc. reduces the modulation accuracy, the available noise floor, and the usability of the generated signals. This issue grows exponentially with modulation bandwidth, so it is sometimes the most critical and costly factor for Vector Signal Generators.

As AWGs grew in bandwidth, linearity, and accuracy, a new approach was possible. Instead of generating the baseband signals, it was possible to generate an already modulated IF signal. The final RF frequency was then achieved through a mixer. Mixers require an additional component to work, a Local Oscillator Generator, and produce two sidebands. Most times one of them must be selected using a suitable BPF. As modulations are implemented mathematically, all the I/Q differential response issues disappear from the equation. However, mixers and L.O. add their own impairments such as intermodulation, conversion losses, flatness, and available modulation BW (i.e. connected to IF frequency).

The continuous advances in DAC and memory technologies have increased bandwidths and sampling rates for AWGs to the 10GHz range and beyond. This allows for the direct generation of modulated RF signals in the UHF, L, S, C and X Bands (fig. 1.1b). This approach can support extremely high modulation BW, well beyond 2GHz, and reduce the complexity and cost while improving flexibility and channel density, which is especially useful for today's radar (i.e. AESA radars) and wireless communication systems (i.e. Massive MIMO). In any case, high-quality direct RF signal generation requires a careful AWG design and waveform calculation. Proper continuous generation of a modulated RF signal requires seamless looping or sequencing of one or multiple waveforms. Modulation signals must be consistent at all levels (symbol, baseband filtering, modulation scheme) when looped and sequenced, so the modulation keeps its integrity and effects like spectral growth are avoided. For direct IF/RF generation, the integrity of the carrier must be kept as well. For a given time window (TW) there must be an integer number of cycles so the signal can be looped without any phase hit. Generally speaking, the number of cycles of the carrier must be an integer. In other words, carrier frequency must be quantized to multiples of 1/TW Hertz. This may be acceptable in some applications but not in others. "True arb" architecture AWGs can change their sampling rate with high resolution and accuracy so the carrier frequency can be adjusted further, by setting a slightly different sampling rate. However, modifying the sampling rate will result in a modification of the modulation signal as well (modulation frequency, baud rate, frequency, and phase deviation, etc). Again, this may be not acceptable in some applications. The timing and frequency accuracy for carrier and modulating signals can be improved by increasing TW (thus the number of samples for the waveform), but this leads to consumption of more waveform memory and increases the calculation and transfer times of those waveforms. An additional issue is the

sampling rate requirements. For baseband signal generation, sampling rate must be higher than the modulation bandwidth (MBW). For direct IF/RF generation, sampling rate must be at least twice Fc + MBW/2, or Fc for small MBW compared to carrier frequency. A modulated RF signal, even for low modulation bandwidths, may require a huge number of samples to keep the required Time Window. Generating the same modulation at a different carrier frequency requires calculating and downloading a new waveform so the new carrier frequency (properly quantized) can be implemented.

Proteus, the new family of high-performance AWG and AWT by Tabor Electronics, has been designed to support the generation and acquisition of high-quality RF and Microwave signals using high bandwidth DACs and ADC (up to 9GS/s and more than 9 GHz usable bandwidth). This document will cover in depth how real-time digital up-conversion (or DUC) and Digital down-conversion (or DDC) is applied to improve the usability, accuracy and RF performance while offering the best-in-class modulation and analysis bandwidths while supporting full coherence and phase control over tens and even hundreds of channels.

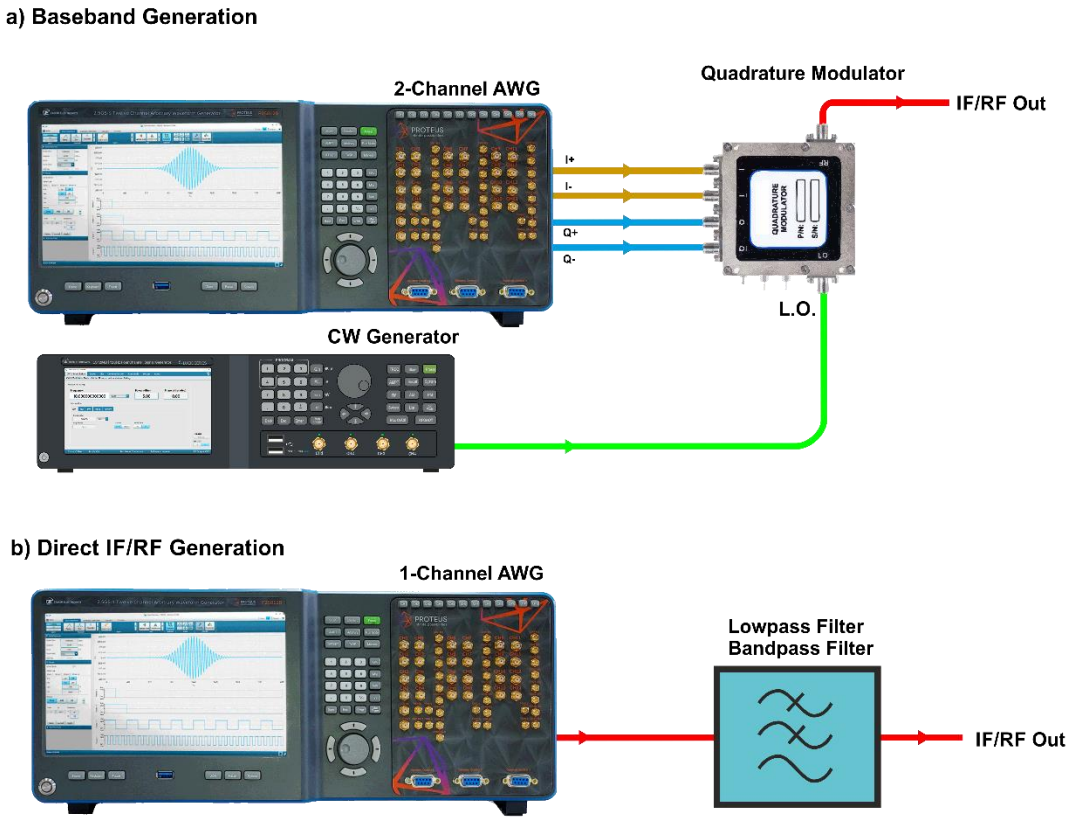


Figure 1.1: Modulated RF signal generation can be performed using AWGs. In a), the traditional IQ baseband signal generation is shown. A two channel AWG generates the two baseband components (as differential outputs in this case) to feed a Quadrature Modulator. This method requires an additional L.O generator to supply the carrier. In b) a high-speed AWG directly generates the modulated carrier using a single channel. There is no need for additional components other than filters and amplifiers.

2 Numerical Up-Conversion Using DUCs

A Quadrature (IQ) Modulator (fig. 2.1) takes a complex baseband signal (In-Phase or real part, I, and Imaginary or quadrature part, Q) and translates it from 0Hz up to Fc, or carrier frequency. Mathematically speaking, it does it by multiplying the complex baseband signal by a complex carrier:

$$SRF(t) = Real\{ (I(t) + j\,Q(t)) \times e^{j\omega t} \} = Real\{(I(t) + j\,Q(t)) \times (\cos \omega t + j \sin \omega t)\}$$

$$SRF(t) = I(t) \times \cos \omega t - Q(t) \times \sin \omega t, \qquad \omega = 2 * \pi * Fc \quad (1)$$

In a traditional IQ modulator, a Local Oscillator produces two sinewaves with a nominal 90º phase difference (quadrature carriers) and those are supplied to two mixers along with the corresponding I and Q baseband signals. Finally, the outputs are combined, and the quadrature modulated RF signal is obtained.

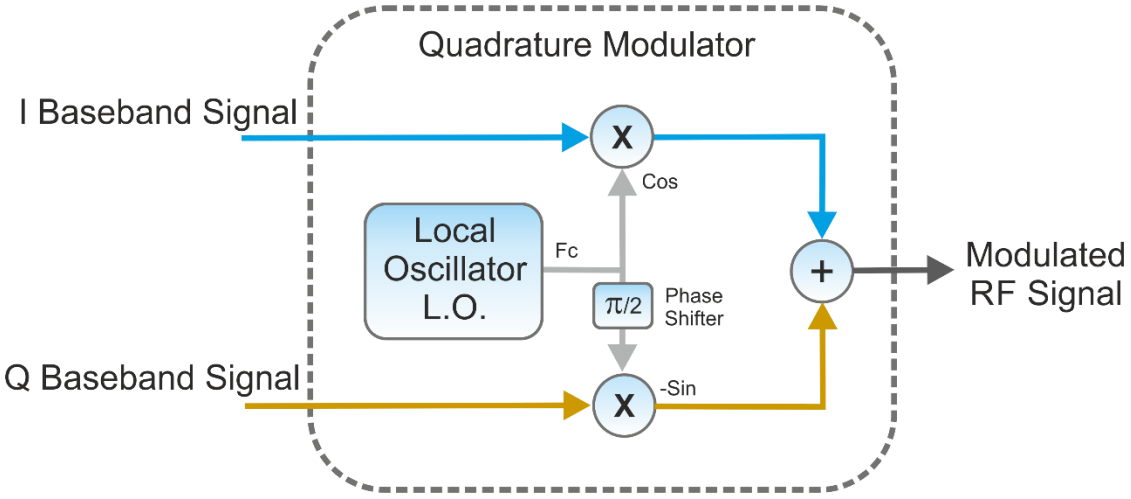


Figure 2.1: Block Diagram of an IQ Modulator. The Local Oscillator (or L.O.) can be external to the modulator itself. The quality of the signal output is influenced by the accuracy and alignment of all the signals and components.

The above process is simple to define in mathematical terms but quite difficult to implement in a practical way, especially when high carrier frequencies and modulation bandwidths are involved. A series of impairments may show up reducing the accuracy and quality of the modulation and the RF signal:

- **Quadrature Imbalance**: It occurs when the I and Q components at the mixer outputs have different amplitudes (fig. 2.2c).
- **Quadrature Error:** This impairment is caused by the lack of orthogonality between the L.O. signals applied to the I and Q mixers respectively (fig. 2.2b).

- **Carrier Feed-Through:** Part of the carrier goes directly, unmodulated, to the final RF signal, interfering with it and wasting power. It can be caused by DC offsets in the I and Q signals, the L.O. signals or by an incorrect working point of the mixers (fig. 2.2d).
- **I/Q Skew:** Differential delay between the I and Q signals becomes more important as modulation bandwidth grows.

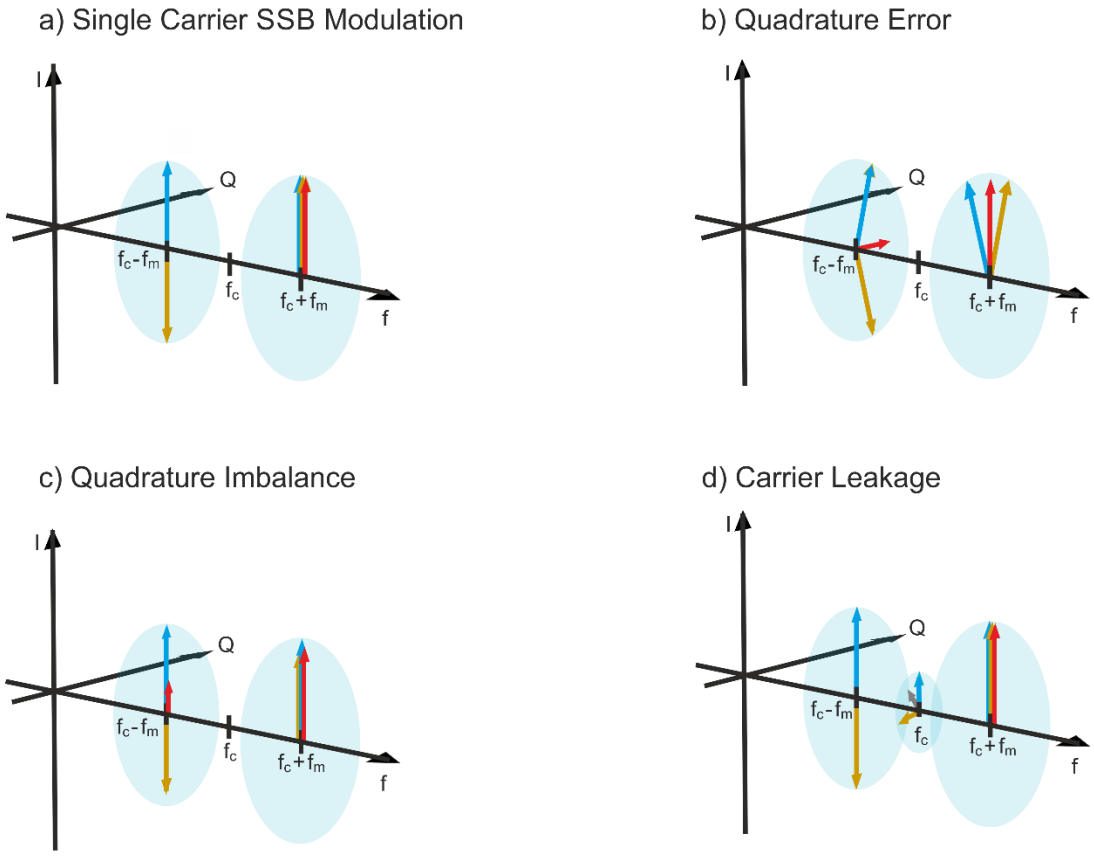


Figure 2.2: IQ Modulation can generate multiple impairments. In this case, a single sideband carrier is generated by supplying two Fm tones with 90º phase. In a perfect modulator, the right sideband is generated while the undesired sideband is nulled (a). If the relative phase of the carriers supplied to each multiplier is not 90º, the quadrature error is produced (b) and an unwanted residual carrier shows up in the opposite sideband. If the amplitude of the I and Q components is not the same, an unwanted sideband shows up as well as the nulling is not complete (c). Finally, any DC component in any of the I or Q components will show up as an unwanted tone at the Fc carrier frequency (d). Real modulators combine all the above impairments that can be a function of the Fm frequency. These are considered linear impairments. Other non-linear impairments are not shown here.

The above impairments, if moderated, can be compensated by a very careful alignment of the modulator and the I and Q signals sources. When the source is an AWG, the I and Q signals can be modified to correct, totally or partially, these impairments. However, both procedures are difficult, and impairments may drift over time, temperature, or frequency so applying them to test equipment, where conditions change from test to test, may be impractical or even not possible.

Direct generation of the modulated RF signal with an AWG removes the above impairments as waveforms are defined mathematically. Even more, impairments may be introduced in a controlled way for margin test purposes with a high level off accuracy and repeatability. Traditional AWGs can generate those signals by playing back waveforms from the waveform memory with the full modulation already implemented in it. As previously mentioned, sampling rate is linked to the carrier frequency more than to the baseband signal bandwidth. Some AWGs, though, can take a different path to solve the IF/RF generation issue. It consists in the implementation of a numerical, real-time IQ modulator, or Digital Up-Converter (DUC, fig. 2.3). In these devices, the waveform memory does not store the modulated RF signal but just the baseband waveforms, either real or complex (I/Q) depending on the modulation scheme. This architecture has important advantages over the traditional direct RF generation using AWGs:

- Carrier frequency is not set by the waveforms stored in the memory and it can be independently set without having to replace the waveforms by operating the digital quadrature L.O. (known as NCO, or Numerically Controlled Oscillator). The carrier frequency is not linked to the time window for the modulating signals anymore.
- As samples stored in the waveform memory carry just the baseband information, bandwidth and sampling rate requirements are set basically by the desired modulation bandwidth. The sampling rate for the baseband waveforms and the final sample rate for the DAC must be adapted, though. This operation can be performed using real-time interpolators.
- As traditional direct RF signal generation, this architecture does not suffer from any impairments as described above.
- Multiple DUC blocks can be combined into a single feed to any of the DACs in the AWG, so more than one carrier with any desired carrier frequency can be generated simultaneously.

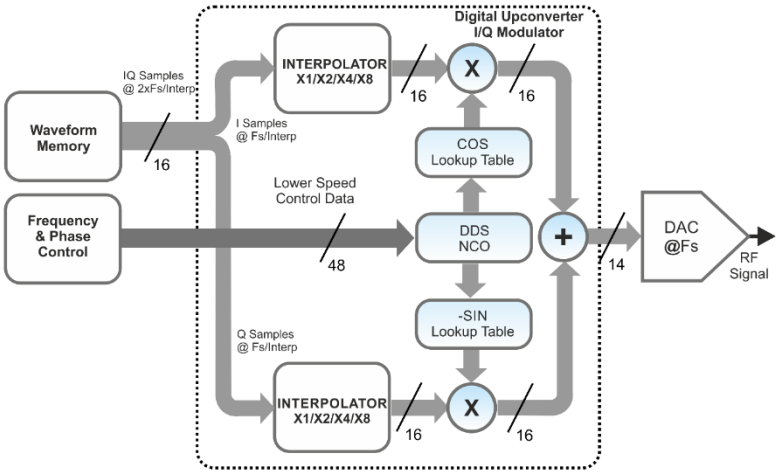


Figure 2.3: Block diagram of the DUC implementation in the Tabor Proteus AWG. Two of these blocks are implemented for each channel. The fully numerical operation removes all the sources for impairments while the usage of interpolators results in an important saving in terms of waveform memory and data transfer rate. The carrier frequency and phase can be changed without recalculating and downloading new waveform data.

The NCO block

A very important component of a DUC is the quadrature NCO (fig 2.4). It can be implemented in different ways so the final carrier frequency is synthesized. Analog L.O. may use a PLL based synthesizer to define carrier frequency. Such synthesizers offer a great deal of flexibility, accuracy, and resolution. However, frequency switching times are influenced by the bandwidth of the closed loop control in the PLL. There may be a trade-off between switching time and phase noise performance. The behavior of the L.O. during the switch may be difficult to predict and random.

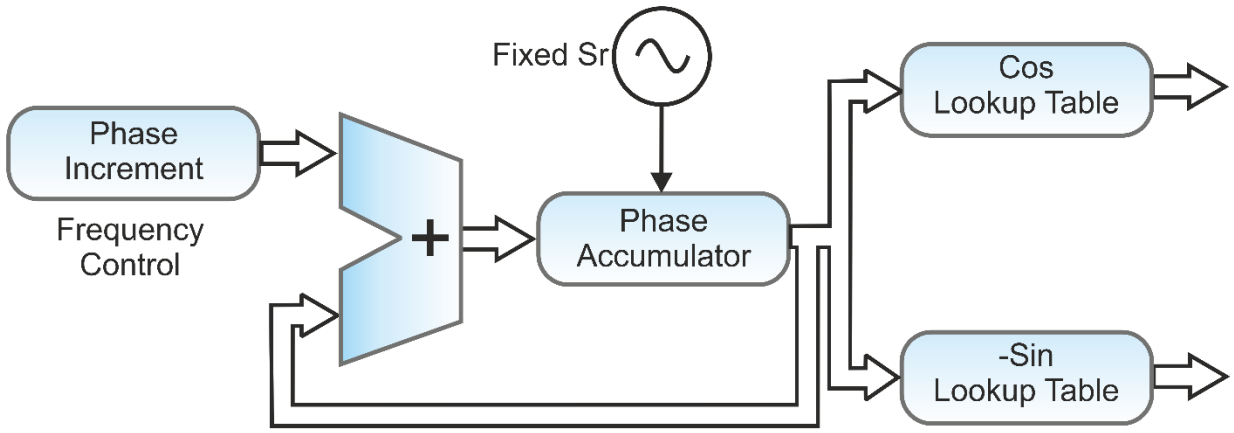


Figure 2.4: A quadrature NCO is the numerical Local Oscillator in a DUC. IQ modulation requires two carriers with a 90º phase difference. In a quadrature NCO, a perfect 90º phase can be obtained by using two lookup tables from the same DDS synthesizer output. Frequency is controlled by the phase increment added for every Sample clock period. Initial phase can be controlled by setting up the initial content of the Phase accumulator. Phase and frequency resolution depend on the size (in bits) of the Phase Accumulator.

Although such a synthesis scheme can be also implemented digitally, NCOs in DUC are typically based on the DDS (Direct Digital Synthesis) architecture. A DDS generates a numerical sinewave by using a phase accumulator and a lookup table. Basically, for every sampling clock, a given number is added to the phase accumulator controlling the frequency. The initial value for the accumulator controls the phase of the sinewave. The value in the accumulator represents the instantaneous phase of the synthesized sinewave, so the corresponding amplitude is read from the lookup table. In a quadrature NCO, two lookup tables are implemented and accessed by the same phase word from the accumulator.

One contains the amplitude values corresponding to the Cos signal and the other one the values corresponding to the -Sin signal. The frequency of the sinewave can be set with a very high resolution according to the size of the phase accumulator. Fc is set according to the following expression:

$$Fc = CW * F_{DAC} / 2^{RES}, \text{ CW = Control Word, RES = size of the accumulator in bits} \quad (2)$$

$$F_{RES} = F_{DAC} / 2^{RES}$$

Using the actual figures from the Tabor Proteus:

RES = 48 bits

FDAC = 9 x 10^9 Hz

Fc = 0 Hz ... 9 x 10^9 Hz

FRES = 9 x 10^9 / 2^{48} = 32 x 10^{-6} = 32 μHz

One of the advantages of NCOs based in the DDS architecture is that frequency switch is instantaneous (from one sample to the next) in a phase-continuous manner so switching glitches are not generated. The size of the lookup tables is limited so some rounding takes place when converting the phase accumulator contents to a given entry in the table. The size (resolution) of the entries themselves in the table is also limited to the size of the multiplier or DAC attached to it. The resolution in the time and amplitude domains of the lookup table is chosen so any impairment (i.e. spurs) introduced by the rounding processes taking place (such as the phase noise coming from the limited number of entries), is negligible in respect to other sources of impairments, such as quantization noise, or the Sampling Clock (Sclk) phase noise.

The frequency of the output sinewaves can be chosen from DC up to the sampling rate. Traditional AWG generation can reproduce signals with frequency components between DC and half the sampling rate (Nyquist Sampling Theorem), called the first Nyquist Zone (or NZ). However, images are produced around multiples of the sampling clock. Each FDAC /2 wide section of the spectrum is called a Nyquist Zone and is numbered depending on its frequency location. Images located at these upper order Nyquist zones can be used, sometimes by filtering out the unwanted images including the one in the NZ #1. As frequency response of the DAC falls with frequency, not all the images can be effectively used for practical purposes. Typically, the second and sometime the third Nyquist Zones can be used if the analog bandwidth of the DAC and the output stage are sufficient, despite the zeroth-order hold response (sin Af / Af with zeros at all multiples of FDAC) of ideal DACs. One way to select the right carrier frequency for the NCO in a higher order Nyquist Zone is selecting the following Fc:

Fc = abs(Fc' – (n – 1) x FDAC), n = NZ #, Fc' = target Carrier Frequency, n >= 2

For even numbered Nyquist zones, the spectrum of the images will be reversed. If the application required preserving the original, non-inverted spectrum, then the complex baseband signal (I/Q) must be replaced by its complex conjugate (by reversing one of the components). However, if the DDS allows for Fc higher than FDAC / 2, it is better to set up that frequency directly in the CW. The subsampling of the NCO output in respect to the target Fc' results in the reversion of the sign of the sin(x) lookup table for the Fc' frequency, so the spectrum around the Fc frequency will be reversed, and the one located in the odd numbered NZs will be right. In this way, baseband data can be preserved unmodified, regardless of the NZ being targeted (fig. 2.5).

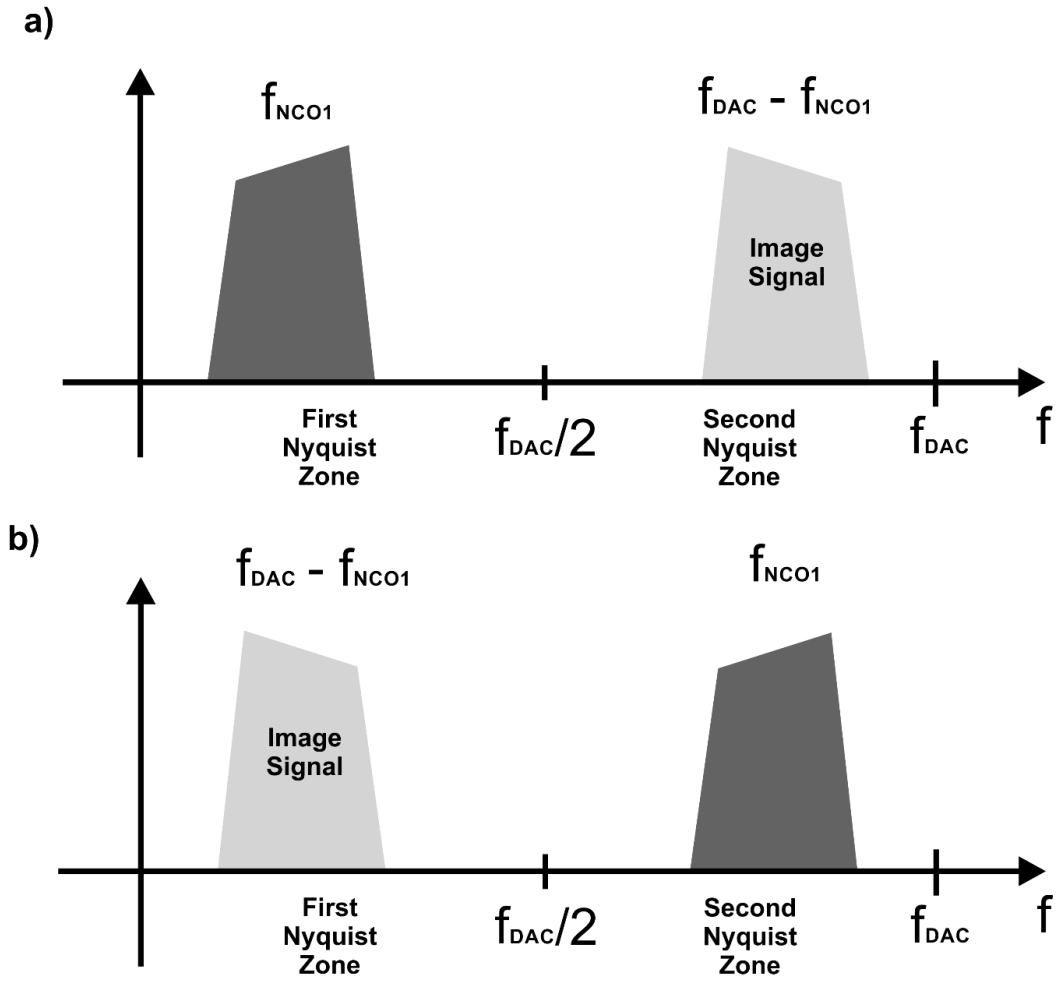


Figure 2.5: The NCO can be set from DC up to the sampling frequency. The resulting modulated signal will include the corresponding images resulting from the sampled nature of the waveforms. When generating a signal in the second Nyquist band, the NCO can be set to the image frequency in the first Nyquist band. However, the spectrum of the modulated signal will be reversed in the second Nyquist band. Although this problem can be fixed by reversing one of the baseband components, it is much better to set-up the NCO frequency at the carrier frequency in the second Nyquist band. In this way there is no need to reverse and update one of the IQ components.

Interpolation

Using the DUC architecture opens the door to separate the sampling rate of the baseband data from the final sampling rate of the DAC. The sampling rate of the complex baseband data must be higher than the modulation BW. A 100MHz modulation bandwidth complex baseband signal could be made of two 50MHz bandwidth signals that should be sampled, at least, at 100MS/s each. However, the DUC must operate at the final DAC sample rate. This means that the I and Q sampled waveforms must be resampled (typically upsampled) before reaching the multipliers. It is extremely convenient for practical purposes that the ratio between the sampling rate of the DAC, and the sampling rate of the baseband

waveform is an integer, N. One simple way to upsample the baseband waveforms could be keeping the same sample value for N samples. However, this method would reproduce the images in the original baseband sample waveforms, and they will show up as unwanted sidebands in the modulated signal. In order to avoid that, a near ideal interpolation process must be applied before reaching the multiplier (fig. 2.6). Ideal interpolation cannot be carried out in the real World, especially if it has to be applied using real-time signal processing. Practical interpolation consists in the upsampling of the incoming signal using a zero-padding process first (by adding N-1 zero samples between actual samples) and then applying a powerful digital low-pass filter using a linear phase response LPF FIR to remove the unwanted sidebands from the interpolated waveform. Practical FIR filters show some roll-off, so the actual modulation BW supported depends on the size of it. 80 to 90% of the theoretical maximum modulation BW are typically supported in real implementations.

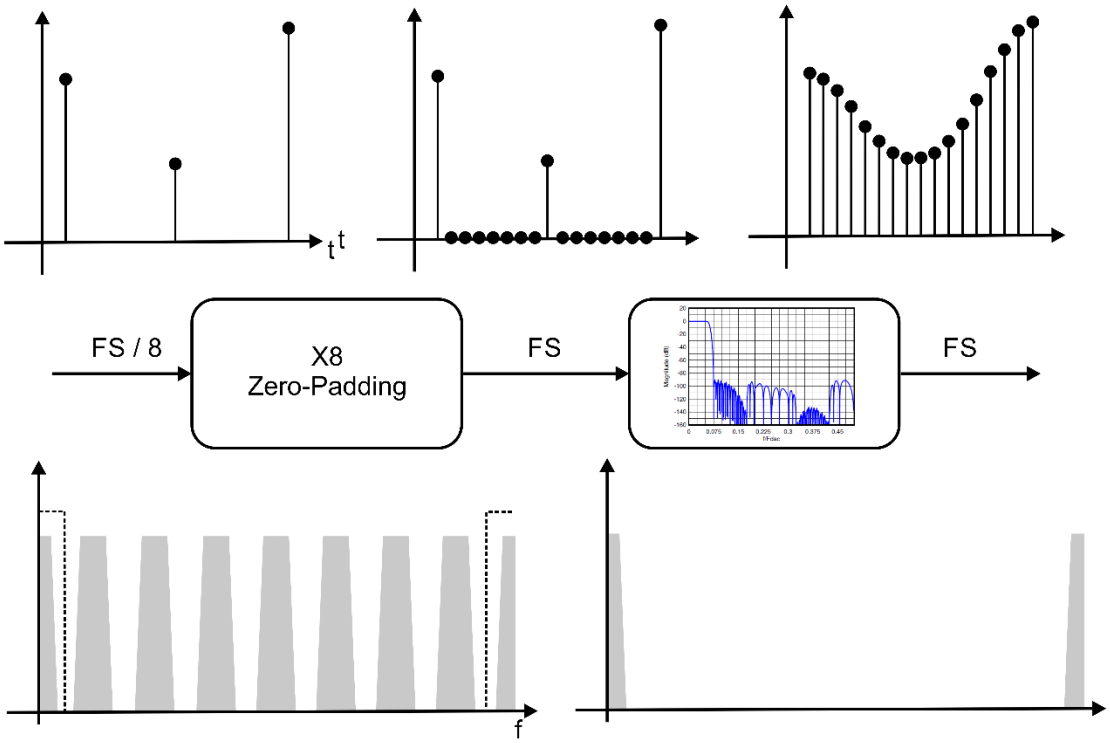


Figure 2.6: Interpolation is a very important factor for Digital Up-Conversion. Interpolators increase the sample rate through a zero-padding process. This process, though, keeps the unwanted images of the signal sampled at a lower speed. A real-time Low-Pass FIR, or interpolation filter, adds the intermediate samples while removing the images above the original first Nyquist zone. Here, the X8 interpolator implemented in the Tabor Proteus AWG is shown.

The Proteus family of AWGs supports multiple interpolation factors (x2, x4, x8) so the sampling rate of the incoming signal can be reduced according to the actual modulation bandwidth requirements and the final FDAC. The FIR filters applied are optimized for each factor as the number of available taps grow with the interpolation factor. Interpolation reduces the size of the waveform in a factor equal to half the oversampling as two samples (I & Q) per sampling period are required.

3 Advantages of DUC for RF signal Generation in AWGs

Waveform Memory Size and Overall Waveform Data Transfer Rate

The gains in terms of waveform memory efficiency when DUC is used to generate RF signals (thanks to the usage of interpolation) has already been mentioned. However, these gains go beyond what can be expected from the mere reduction of the incoming sample rate for baseband waveforms. Generating accurate RF signals through direct generation of the carrier is not as straight forward as it could seem. For a continuous modulation, the waveform must be calculated in such a way it can be looped seamlessly. This requires an integer number of symbols, an integer number of carrier cycles, and an integer number of samples. Some modulation schemes may require the number of symbols in the sequence, to be a precise number in order to be meaningful for the receiver under test. Additionally, the waveform length must be always a multiple of a given number in high-speed arbs as samples are read in parallel from the DDR massive memory. The above considerations may result in the need to round the actual symbol rate or carrier frequency to the closest value resulting in the required waveform continuity conditions. An example can help to understand this issue. Let's take a DVB-T signal (8MHz channel BW, symbol duration 924μs for 1/32 guard interval) for basic receiver test at UHF channel #69 (858MHz). A minimum consistent DVB-T signal, so the receiver can recognize the modulation parameters, requires a complete sequence of TPS carriers (these carriers supply the modulation parameters for the DVB-T signal), made of 32 OFDM symbols. Let's generate such a signal through direct generation of the modulated RF signal, at 9GS/s with an AWG with a 64 samples waveform length granularity. The first thing to do is calculate the duration of the 32-symbol sequence:

Time Window (TW) = 32 * 924μs = 29.568ms

The corresponding waveform length can be calculated:

Waveform Length (WL) = SR * TW = 266,112,000 samples

Fortunately, this number is already an integer and a multiple of 64 so the length does not have to be rounded to the nearest integer multiple of 64 (that should change the accuracy of the symbol rate up to 0.12 ppm), or repeated in memory until an integer multiple of 64 would be obtained (no symbol rate error in this case, worst case would lead to repeating the same sequence of samples up to 64 times, so more than 17G Samples would be required).

Next step is calculating the right carrier frequency so an integer number of cycles for the carrier will be obtained:

Number of Carrier Cycles = TW * CF = 25,369,344

Again, the number of cycles at the target carrier frequency is an integer so the carrier frequency will be accurately generated. If this is not the case, adjusting the number of cycles to the nearest integer could result in a 17Hz error for the frequency carrier.

Let's take now the case of the same AWG using a DUC with 8X interpolation. This interpolation results in a baseband sample rate of 1.125GHz so modulation BW is around 1GHz, more than enough for this DVB-T signal. Calculations must be repeated for the new conditions:

Waveform Length (WL) = 33,264,000 samples

This is an integer number multiple of 32 (granularity is halved for complex signals stored as interleaved IQ pairs) so there is no need to round the number or repeat the sequence within the waveform memory. As the NCO frequency is what determines the carrier frequency, the only important consideration is the frequency resolution of the NCO, which is typically in the tens of μHz range. The gain in terms of waveform memory usage is a factor of four (complex samples are made of two real samples each). Even more, any FC error coming from a frequency error in the sampling clock can be corrected by setting up a corrected FC in the NCO. The only way to do so is by modifying the sampling rate itself.

Carrier Coherence

The above considerations are also important for pulsed signals (i.e. radar) if carrier coherence must be maintained between RF bursts (fig. 3.1). Keeping carrier coherence is important in multiple applications. Coherence requires preserving frequency and phase during all the testing time. Using direct generation of the RF carrier (no DUC), this may be difficult, if not impossible, depending on the signals being generated. In a WiFi sequence of packets being made of waveforms segments of different lengths, the number of cycles of the RF carrier for all the RF bursts might not be an integer number for all of them. As these signals are bursts, it looks like keeping the number of cycles being an integer number may not be necessary. However, a careful analysis shows that the phase of the carrier will change from segment to segments, which is not acceptable for some applications. The NCOs in the DUC keep going independently of the waveforms being sequenced, so the right coherent phase is maintained, as long as the NCO is not reset. Another situation, where direct carrier generation may result in the loss of carrier coherence, is when segment generation is asynchronously started through software, or hardware trigger events. Again, the NCO independence of the waveform memory reading process, makes coherence possible no matter the way waveform segments are triggered or sequenced.

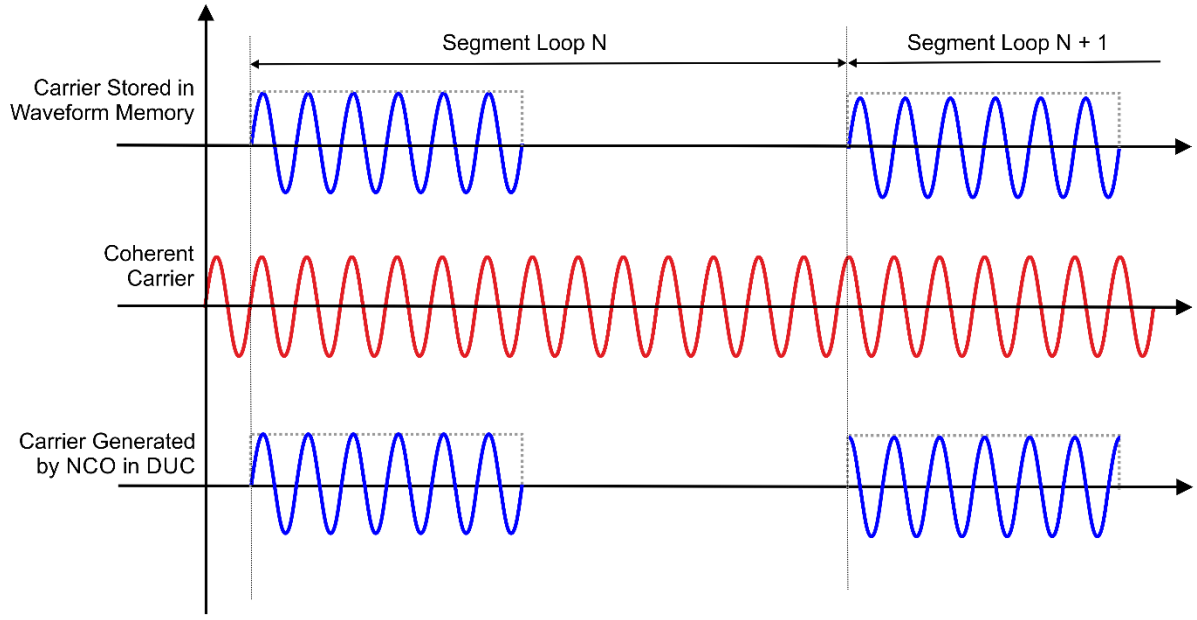


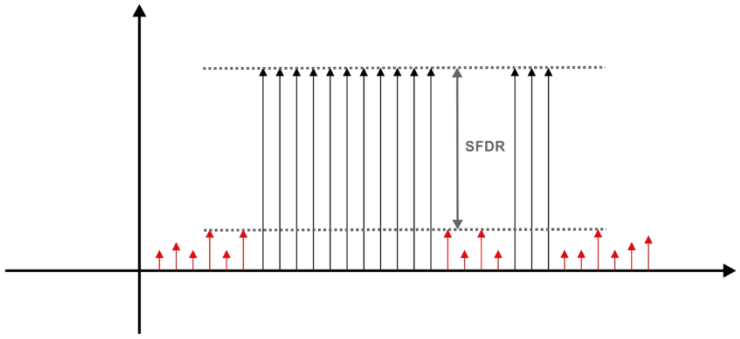
Figure 3.1: Many applications require keeping the coherence of the carrier indefinitely. Direct generation of the carrier embedded in the waveform does not guarantee coherence unless the carrier frequency is limited to one resulting in an integer number of cycles within a segment. Even in this case, coherence is only kept when segments are seamlessly generated. For asynchronous generation (i.e. after some external trigger event) coherence will be lost as seen at the top. As NCOs in DUCs run independently of the modulating waveforms (doted pulses), coherence is kept no matter what, as seen in the bottom trace.

Quantization Noise Dithering

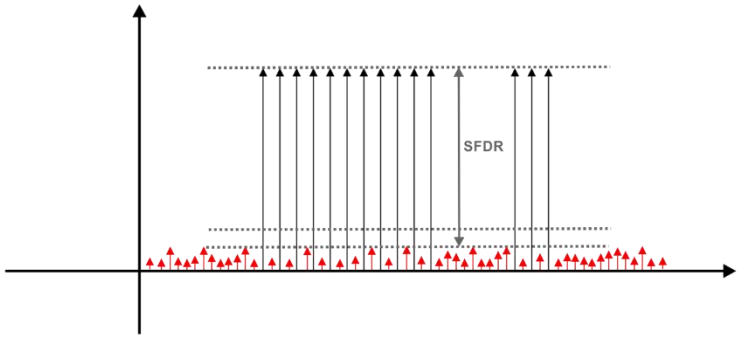
When AWGs generate continuous RF (or non-RF) waveforms by looping the same segment over and over again, an interesting effect occurs (fig. 3.2). Quantization noise, generated even by perfect DACs and seen as a random process when dealing with real-world signals, becomes periodic. Quantization noise can be modelled as a constant distribution white noise with one LSB peak-to-peak amplitude. Ideally, the SQNR (Signal-to-Quantization-Noise Ratio) for a sinewave using the full DAC range depends on the resolution in bits of the DAC:

SQNR(dB) = 6.02 x N + 1.76, N = DAC resolution in bits

a) Multi-Tone Generation: One repetition period (k = 1)



b) Multi-Tone Generation: Two repetition periods (k = 2)



c) Multi-Tone Generation: Internal DUC

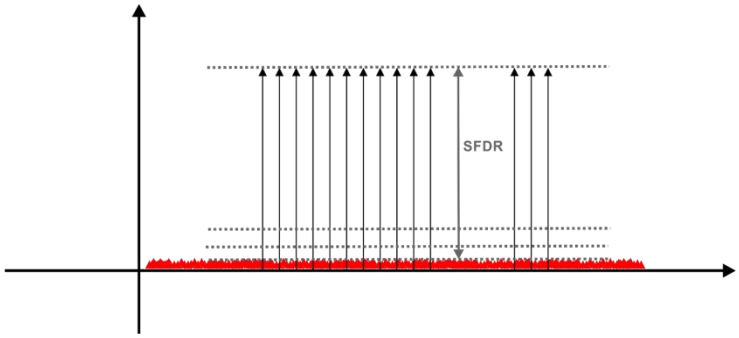


Figure 3.2: When looping a waveform, such as a multi-tone signal, quantization becomes periodical, and it shows as a series of discrete tones (a). If the waveform length holds multiple cycles of the waveform without repeating the same sample sequence, the repetition period grows, and the average power of the tones is reduced at the price of a longer waveform (b). As carriers generated by the DUC does not have to be synchronous with the waveform, the repetition period of the waveform can be extended to hours, days or weeks. As a result, quantization noise becomes denser and the best possible SFDR is accomplished (c).

The fact that quantization noise becomes periodic, has a major impact on the spectrum of that noise. While quantization noise in operating devices (such as a CD player or a Wi-Fi transmitter) can be modelled as random, so its spectrum is dense and evenly distributed, quantization noise for repeating waveforms shows up as discrete spectral lines located at multiples of the repeating frequency). The average power level of these discrete tones depends on their number, as total power remains constant,

as depicted in the expression above. In other words, the shorter the signal, the bigger the distance between those quantization noise tones, and the higher the average power of them. Eventually, these tones can show up over the background noise and be a major contributor to the reduction of the SFDR performance, modulation quality degradation, and ACPR. Just to show an example of this, let's consider a satellite link QPSK signal at 25.776MBaud (51.552Mbps) generating a PRBS7 test sequence. As a PRBS7 sequence is made of 2^7-1 = 127 bits, the same sequence of bits must be repeated twice to fit an integer number of QPSK symbols (2 bits /s symbol). The minimum sequence of symbols will be then 127. This means that the minimum TW for this signal will be:

$$TW = 127 / 25.776 \times 10^6 = 4.927\mu s$$

This will result in a repetition rate of 203KHz and, therefore, quantization noise will show up as harmonics of that frequency. One way to reduce the average power of these tones is by increasing the sampling rate when possible, as the noise will be spread over a higher BW. When this is not feasible, it is possible to reduce the average power by reducing the repetition rate. This cannot be done by simply appending multiple copies of exactly the same waveform in the memory, because this will not change periodicity. There are two ways to handle this situation when direct RF carrier generation is involved:

1. Calculating a new waveform where the multiple repetitions of the same basic waveform are not sampled in the very same sampling instants. A way to make sure this happens would be selecting a waveform length, which does not have any common divide with the number of symbols in a basic sequence. This way, the signal will not repeat exactly in the same way within the waveform, and the noise will be spread over a larger number of tones.
2. The second technique is dithering. In this scheme, the same basic sequence of samples is repeated, and then a random number (1/2 quantization level peak-to-peak amplitude is enough) for all the samples. As a consequence, quantization noise will not repeat until the complete segment is looped, and the average level of the quantization tones will be reduced, at the expense of increasing the overall noise in the signal.

Procedure #1 is better as it does not increase the noise power in the system. The best way to proceed is selecting the number of repetitions of the same symbols sequence to be a prime number, and then calculate a suitable sampling rate and waveform length so the latest is not a multiple of the prime number. In this way, no exact repetitions of the same sample sequence will occur in the segment, and the repetition rate for the quantization noise will be reduced by the same prime number factor. If we apply this approach to the AWG used in the previous example and the DVB-S test signal mentioned above, we can calculate the generation parameters for the maximum FDAC = 9GS/s. The TW for one consistent sequence of symbols (127 QPSK symbols) is around 4.927uS. If we use the exact numbers and the target sample rate for 101 repetitions (prime number), the waveform length will be:

WL = 4,478,701 samples

We will use the closest multiple of 64 (the WL granularity for this AWG) lower than the above WL:

WL' = 4,478,656 samples

Selecting a lower number is convenient as the Sampling rate can be reduced a little bit (from 9GHz down to 8.999909572 GHz) to keep the right baud rate. Reducing the effects of quantization noise in this signal, when using an AWG equipped with a DUC, is much easier as the waveform going to the DAC is not the one in the waveform memory, but the mixing of it with the real-time quadrature sinewaves being generated by the NCO. If the frequency set in the NCO is not an exact integer multiple of the repetition frequency of the sequence in the waveform memory, the sequence will not be the same all the time and, as a consequence, quantization noise will be spread densely over the full spectrum and no noise tones will be visible. Repetition period for the output samples after the DUC block can range

from seconds until weeks, so at the operational level, it will behave as an ideal white noise without increasing the overall noise level. In this case, it means that the WL could be kept to the minimum 44,288 samples.

Processing Gain, Effective Bits, and DAC Modes

Probably, the most important bottleneck for high-speed arbs is the DRAM to DAC interfacing. Even when using massive parallelization, the sustained transfer rate to the DAC is limited. This results sometimes in a trade-off between DAC resolution and sample rate as the product is the sustained data rate between memory and DAC. For a 9GS/s, 8-bit resolution DAC, transfer rate is 9GByte/s. For a 2.5GS/s, 16-bit DAC, data rate will be 5GByte/s. And sometimes this is not the full picture, as some other information may be transferred concurrently from the waveform memory such as markers, so some instruments may lose resolution when markers are activated, as transfer rate has reached the maximum allowed by the implemented architecture.

DUCs also have an impact on this issue as they incorporate interpolators. When DUCs are implemented right in the DAC block, waveform data being transferred to the DAC block is reduced by half of the interpolation factor (for IQ modulation). If the waveform data transfer rate for a 9GS/s DAC is limited to 5GByte/s it is possible to transfer up to 1.25GSample/s 16-bit IQ sample pairs (so Modulation BW goes beyond 1GHz). The DUC using an 8x interpolation factor can handle such rates when sample rate is 9GS/s, while direct generation of the RF carrier would be limited to 8-bit samples over 2.5GS/s. Interpolation opens the door to use higher than necessary sampling rates and this results in what is called "processing gain". Basically, oversampling a waveform by a factor of 4, is like using a DAC with one additional bit of resolution at the original sampling rate (fig. 3.3).

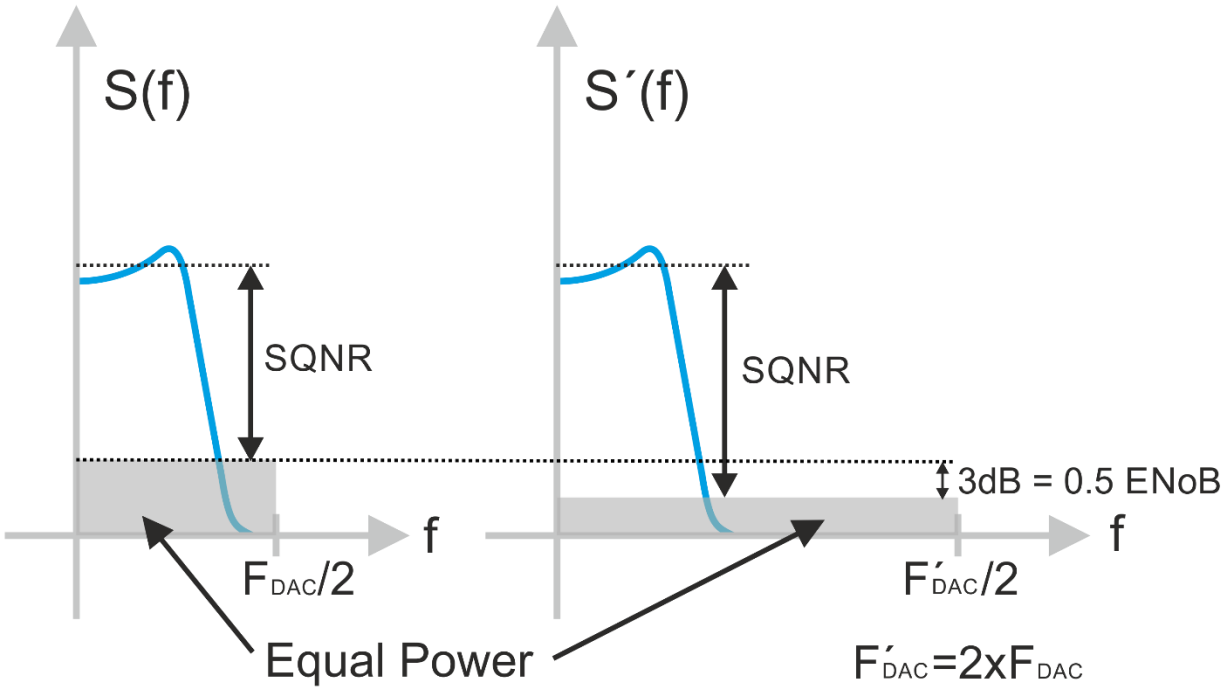


Figure 3.3: Near-Ideal Interpolation (Oversampling) is part of digital up-conversion. Oversampling results in a reduction of the quantization noise power spectral density as the same. Increasing sample rate by a factor of two results in a 3dB reduction in the noise power over the signal's BW, so it is like increasing effective number of bits (ENoB) by 0.5.

Simultaneous RF and non-RF signals synchronous generation (Envelope Tracking)

AWGs are general purpose signal generation devices. The DUC mode can be used to generate RF signals conveniently, but the same device must be capable of generating signals through direct conversion, so non-RF signals with DC components can be generated as well. Some applications may require the synchronous generation of both kinds of signals (i.e. envelope tracking or Qubit Control, fig. 3.4). In some AWGs, though, the DUC mode can be activated or deactivated for all the channels simultaneously. Fortunately, it is possible to generate baseband (non-RF) signals through the DUC as well. The procedure is quite simple as it requires setting both the frequency and the initial phase for the NCO to zero, and then use the I samples (Q samples can be set to "all zeros" to reduce digital noise) as the non-RF signal to be generated. As the waveform will go through the same processing blocks in the DUC (oversampling, low-pass filtering, etc.) the synchronization and sampling rate for all the signals, RF and non-RF, will be consistent.

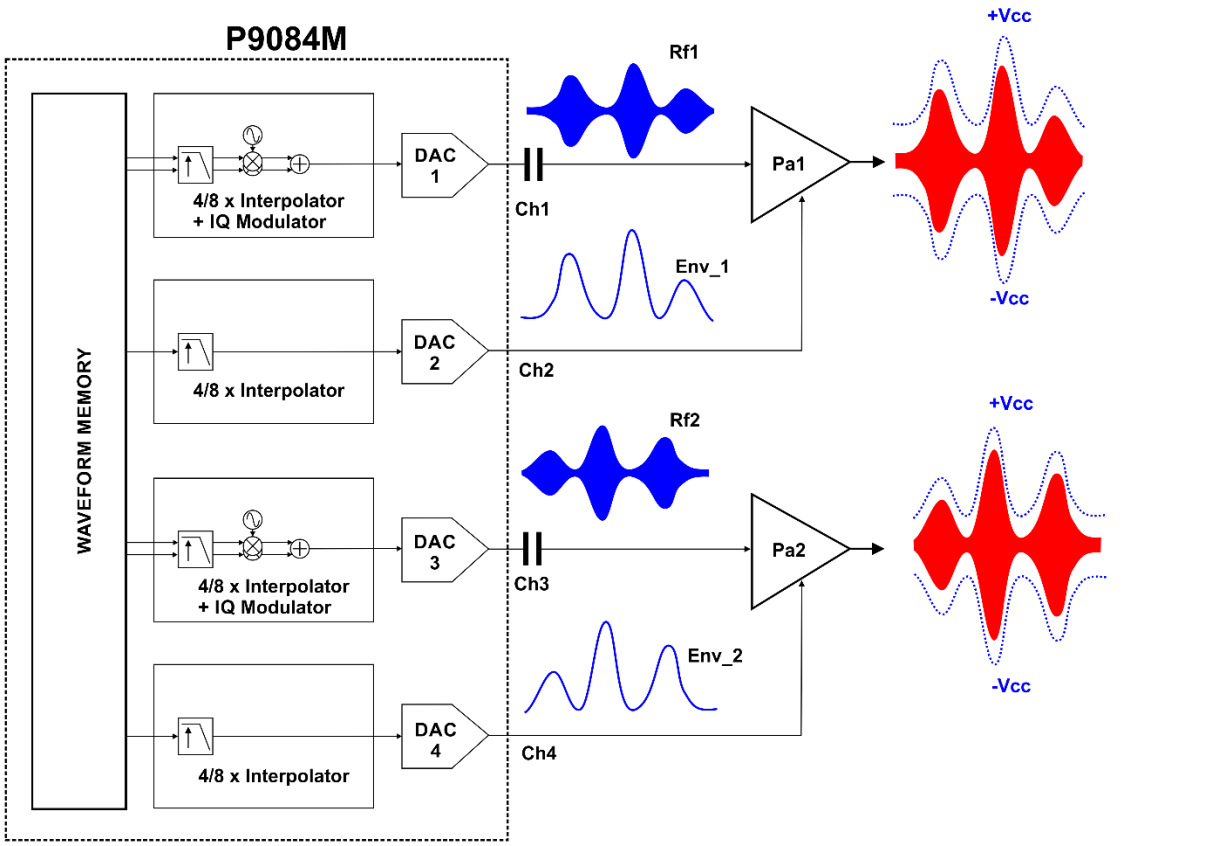


Figure 3.4: AWGs are a very versatile tool as they can generate any kind of signal. In particular, they can generate RF and baseband signals simultaneously. The Tabor P9484M is a good example. Here two channels generate two different AC-Coupled RF signals, and the other two channels generate the corresponding synchronous DC-Coupled "envelope tracking" signals to properly handle a high-efficiency RF Power Amplifier.

Waveform normalization and quantization for DUCs

Baseband IQ signals must be calculated, processed, and transferred in order to generate valid RF signals using a DUC. Especial care must be taken when building those waveforms. First, although the I and Q waveforms can be handled as a pair of waveforms, it is important to keep in mind that those are, in fact, just a single waveform made of complex numbers, and it must be handled in that way. One important issue in AWGs is obtaining the maximum SNR without distorting the signal. This is even more important for RF signals. The usual practice is using the full DAC range, so waveforms are normalized to that range and then the required amplitude and DC level is set using the output voltage and offset controls (fig. 3.5). This is also true when using the DUC. However, the full range of the DAC is now connected to magnitude of the complex signal and not to the amplitude of each of the real and imaginary (or I and Q) components. Normalization to the DAC range must be carried out over the peak magnitude of the complex signal so both signals must be normalized together.

Another issue is the DAC range itself. The DUC interprets the midrange level as 0.0 so the available DAC range goes from 1 (instead of 0) up to $2^N - 1$. Using the "all zeros" level as the extreme value would result in a small DC offset in both the I and Q components (fig. 3.6). This DC offset will show up as a small residual carrier impairment (visible in a Spectrum Analyzer even for a 16-bit sample). It will also

result in a residual RF carrier generated between RF pulses or bursts. Keep in mind that one of the advantages of the DUC architecture is that residual carrier can be avoided and that the “OFF” state for pulsed RF is perfect.

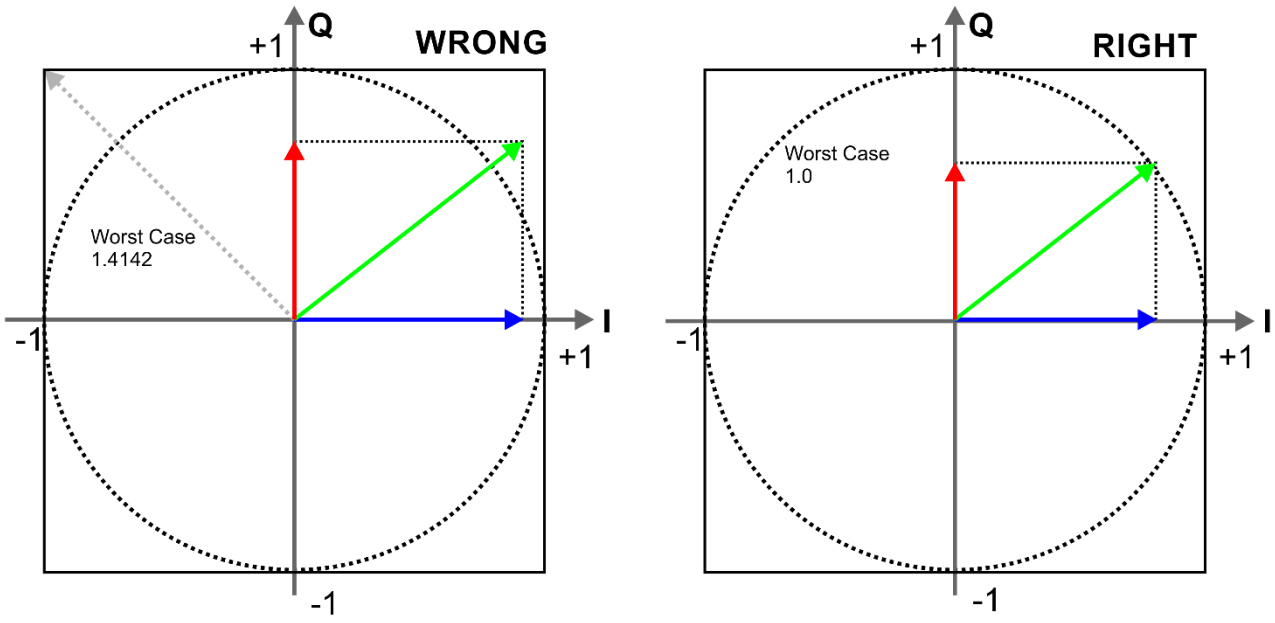


Figure 3.5: When using DUCs in AWGs, I and Q samples are stored in the waveform memory. I and Q samples are combined through the IQ modulator. It is important to avoid DAC clipping as this results in a heavy non-linear distortion, spectral growth, and poor modulation quality. In order to avoid clipping, the I and Q waveforms must be normalized in a way that the maximum peak is always below the DAC range.

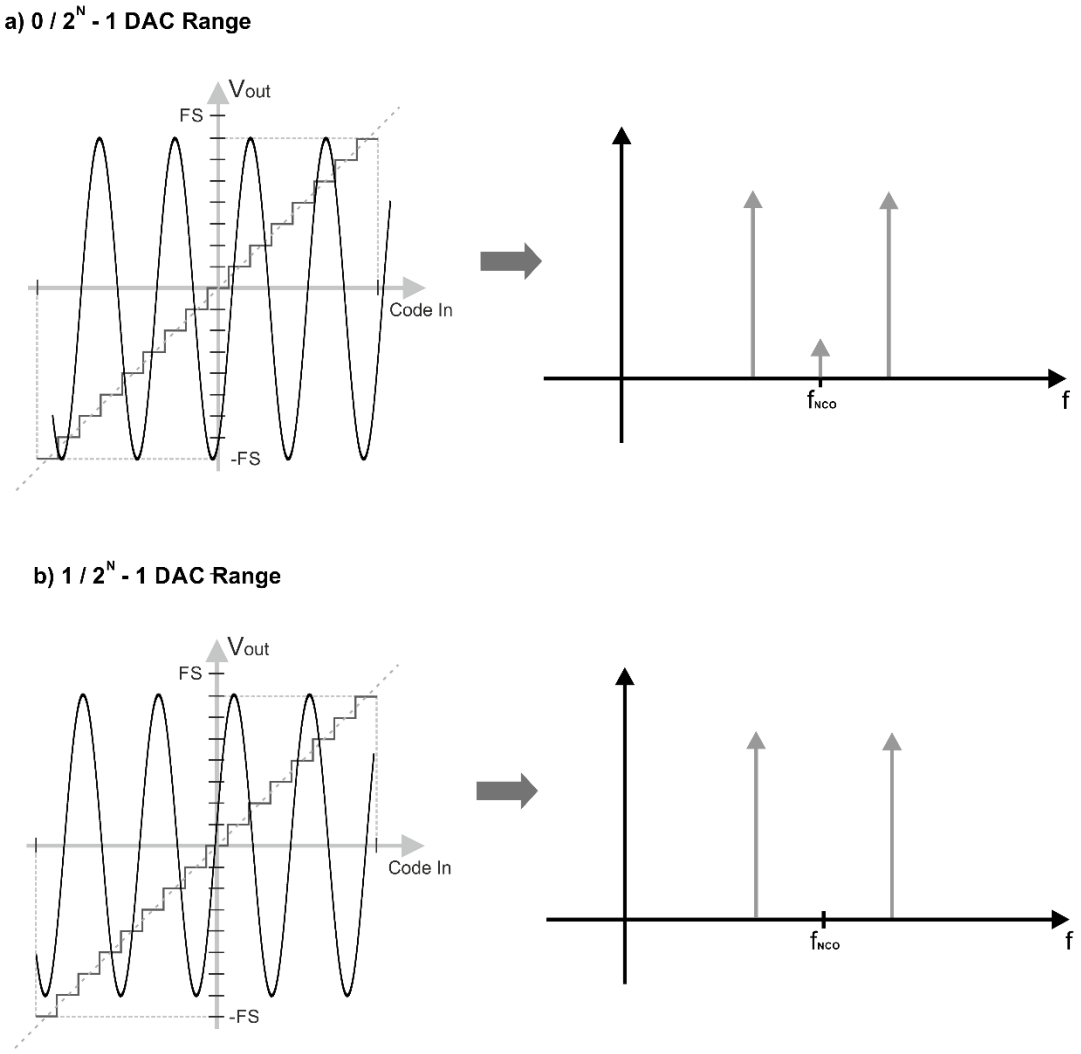


Figure 3.6: The best way to leverage the full dynamic range of a DAC is by normalizing the range of the waveform so it covers all the quantization levels of the DAC (a). This approach is not perfect when waveform data is normalized for DUC use. It is important to keep perfect symmetry around the "zero level (2^{N-1} level) so the "all zeros" level (the minimum) must be avoided in order to remove an unwanted residual carrier (b).

Generating Multiple Modulated Carries Through a DUC

Most times DUC are used to generate a single modulated carrier and the carrier frequency is set only by the NCO settings. As previously mentioned, traditional analog IQ modulators are difficult to align, and they generate multiple impairments. Many of these impairments generate noise (or self-interference) within the BW Occupied by the signal. Quadrature errors result in unwanted images (frequency components in one of the sidebands generate interfering signals in the opposite sideband) and components such as carrier feed-through (therefore some OFDM-based standards do not use the

central carriers). One way to minimize the effects of those impairments is by shifting the complex baseband signals (fig. 3.7) by rotating them so the final I' and Q' waveforms are calculated as follows:

$$I' = I \times \cos(2\pi F_S t) - Q \times \sin(2\pi F_S t)$$

$$Q' = I \times \sin(2\pi F_S t) + Q \times \cos(2\pi F_S t)$$

F_S can be positive (so the $Fc > F_{NCO}$) or negative (so the $Fc < F_{NCO}$). If the different F_S are properly selected, signals will fit in the available modu higher than half of the modulated waveform BW, no image will overlap and the carrier feed-through will be out of the useful signal. This methodology can be used to combine multiple IQ modulated baseband waveforms so the DUC can generate multiple, independent modulated signals over the available Modulation BW. There is no need to use these tricks to avoid such impairments in DUCs as all the modulation process is numerical, and impairments such as quadrature error and imbalance, and carrier feed-through are, by definition, non-existing. However, when an external IQ modulator is necessary (i.e. to reach higher carrier frequencies), the rotating I and Q components can be generated using a two-channel AWG equipped with built-in DUCs. To do so, just set the two channels to work in the regular DUC mode and set the I component for channel 1 with the I component of the complex signal and the Q component to "all zeros". Then set the Q component of the complex signal of channel 2 with the Q component of the complex signal and the I component to "all zeros". Next set the same frequency and phase for each NCO. Frequency must be set to the desired positive frequency shift. For negative shifts, just swap the target for the I and Q components (or the target component for each channel). This arrangement only works if all the NCOs are phase coherent.

The same approach can be used by numerical IQ modulators (or DUC) so IQ signals may be shifted in frequency by rotating the waveform data being downloaded to the waveform memory. As there is no need to do so to avoid IQ modulation impairments in DUCs, the only reason to do so is generating multiple RF signals at the same time within the modulation BW of the DUC (i.e. multi-tone signals). Keep in mind, though, in this case the available DAC range must be shared by all the RF signals, so average power will be reduced as the number of signals grow.

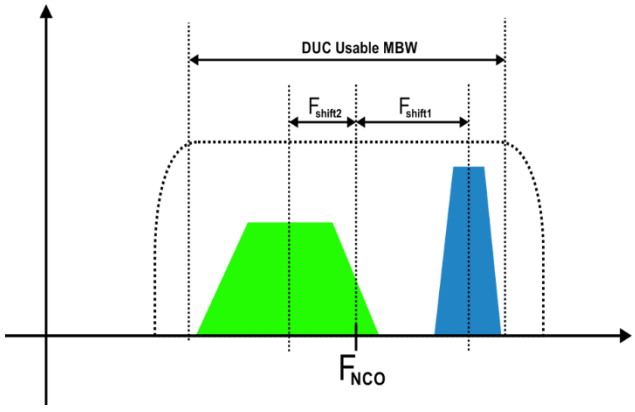


Figure 3.7: Multiple independent, modulated signals can be generated using a single DUC if the full bandwidth of the combined signals fits in the available Modulation Bandwidth (MBW) of the IQ modulator.

4 Implementation of the DUC in the Proteus Family

Block Diagram

The Proteus family of products (fig. 4.1) incorporates DUC in the P258X (optional) and P948X products, regardless of the platform (B, D, or M). The main differences between the P258X and P948X are maximum sample rate (2.5GS/s vs. 9GS/s) and the 8-bit DAC mode available in the P948X products so direct generation without interpolation or digital up-conversion is possible up to 9GS/s. The PXIe modules can incorporate two or four channels. Channels are grouped in pairs (ch1&ch2, ch3&ch4) so two channel instruments incorporate one pair while four channel instruments incorporate two pairs. Each pair shares the same dynamic memory bank, so the connection is shared among the channels. The overall maximum transfer rate for each pair is 10GBytes/s. This means that 16-bit samples can be transferred to all channels up to 2.5GS/s while the 8-bit mode allows for 9GS/s transfer to one of the channels in the pair (Ch1 for the first pair and Ch3 for the second pair).

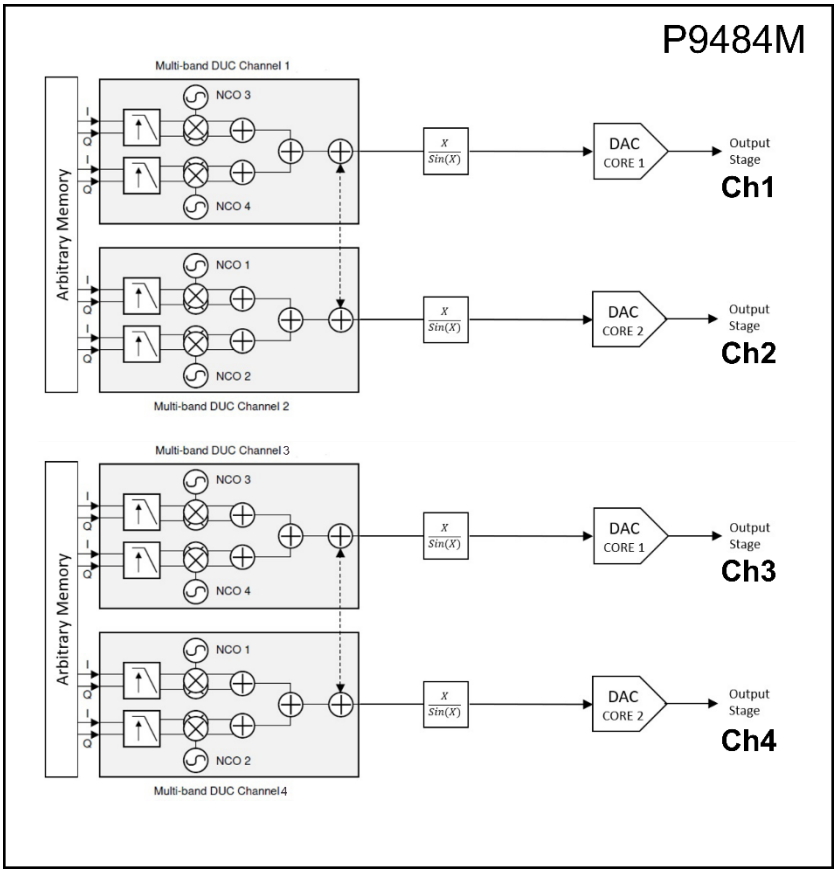


Figure 4.1: Block diagram of the Tabor Proteus P2584M and P9484M when used in the DUC mode. Notice the availability of two independent DUCs per channel. Each DUC can be fed with their one IQ baseband waveforms. There are several IQ modes using the available DUCs in different ways to offer more carriers per channel or more modulation bandwidth.

DUC Modes

When it comes to the DUC, the block diagram shows that there are two independent DUCs for each channel (fig. 4.2). Each DUC incorporates its own NCO so the carrier frequency can be set to different frequencies all over the tuning frequency range (DC up to the current sample rate). Interpolators can implement interpolation factors 1x, 2x, 4x, and 8x. One (ONE Mode) or both (TWO mode) DUCs can be used at a given moment. However, the maximum sample rate and modulation BW depends on the interpolation factor and the number of DUCs being used. Additionally, there is a switchable connection between the output of the DUC block in one of the channels of each pair and an adder connected to the output of the other channel of the same pair, so the combination can be fed to the corresponding DAC while the other DAC remains inactive (HALF Mode). In this way, modulation bandwidth and data rate can be doubled as I data is fed only to the DUC of one of the channels while Q data is fed to the other. In this mode, just the I path for each DUCs is used while the two NCOs are set to the same frequency, while the relative phase is set to 90o. In other words, each DUC block generates half of the IQ modulation.

The processing chain in the DUC uses 16-bit integer arithmetic and the DUC only works in the 16-bit mode, so all the baseband waveforms are defined as a set of 16-bit IQ pairs. The resolution of the DAC itself is 14-bit. It is important to use a higher resolution for samples and all calculations in order to keep calculation noise (coming from integer arithmetic rounding in the interpolation FIR, multiplier, and adders) below the resolution of the DAC, so the quality and RF performance of the final signal is not degraded. FIR filters in the interpolator are optimized for usable bandwidth, flatness, and stop-band attenuation. The number of coefficients depends on the interpolation factor being applied. The filter roll-off is designed to maximize the usable bandwidth so the maximum attenuation is not reached under the Nyquist frequency for the waveform before interpolation. Instead, the maximum attenuation band starts close to the image frequency of the maximum frequency of the flat-response band. It is necessary, then, to make sure that the maximum frequency component of the waveform before interpolation is not larger than this frequency.

There is a numerical 6dB independently switchable attenuator at the output of each one of the two DUCs in the Proteus' DUC block. The main purpose of this attenuator is avoiding clipping when the DUC is operated in the TWO mode. As two IQ modulated signals are added together and NCOs are independent, the worst peak for the combined RF signal will be twice the one for each of the component RF signals. If both IQ waveform are normalized for the maximum DAC range, attenuating both signals by 6dB before adding them, will avoid any chance of clipping the DAC. At first sight, the same result could be obtained by dividing all the I and Q samples by two. However, although this method would avoid clipping as well, the effective resolution of the baseband data would be reduced to 15-bits, and calculation noise would be noticeably higher than calculating each RF signal with 16-bit, and then rounding the result to 15-bit (dividing by two) before the adder. This is a simple but effective approach when peak power is the same for both RF signals. However, when power (or peak-to-peak amplitude is different), then a joint normalization may be better. Keep in mind that sampling for both IQ signals (and waveform length as well) is the same for both DUC in the TWO mode. The normalization procedure must find the maximum

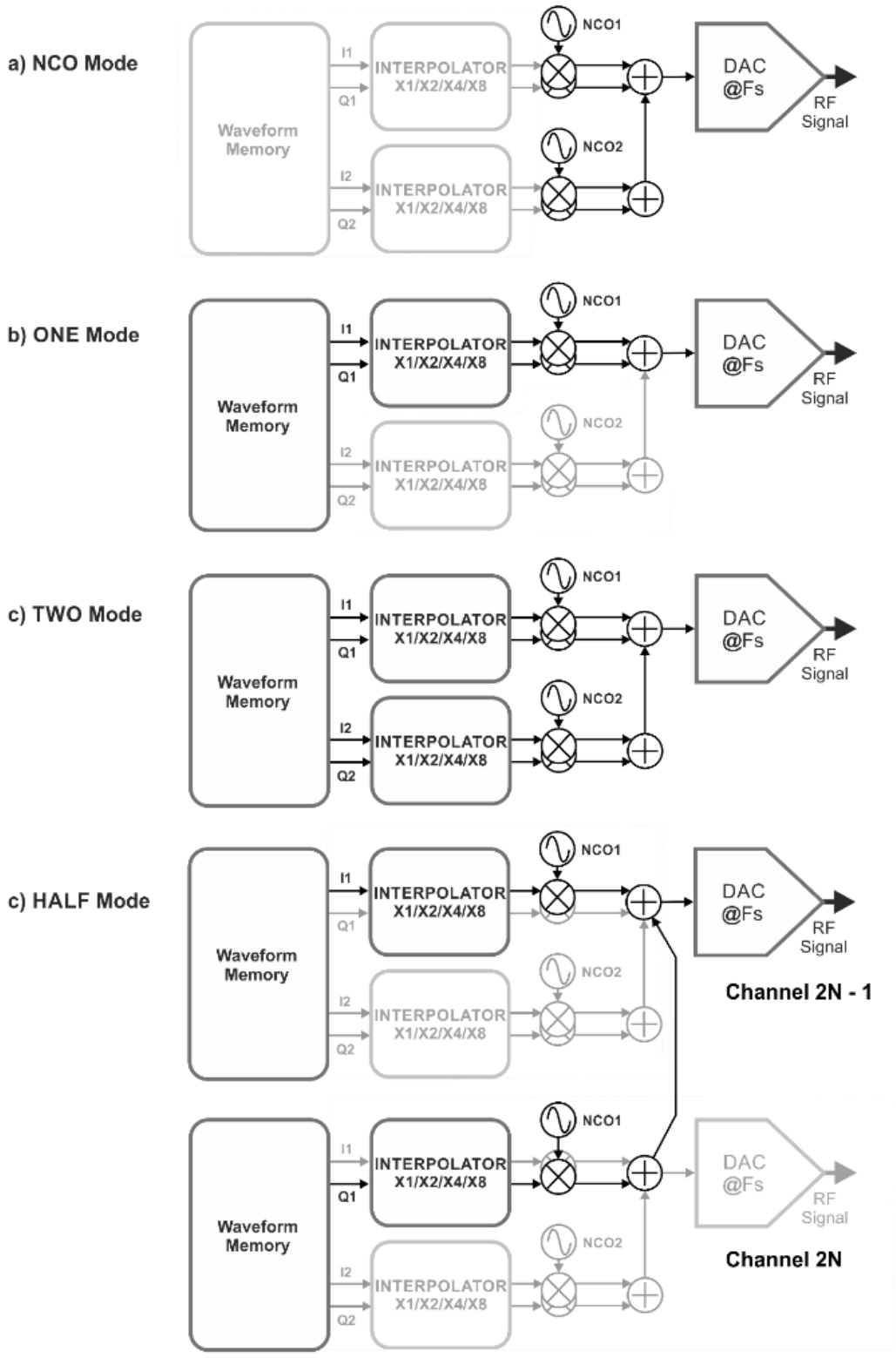


Figure 4.2: There are 4 DUC modes in the Tabor Proteus product. One of them, the NCO mode (a), uses the internal NCOs to generate sinewaves with controlled frequency and phase. The ONE mode (b) uses just one of the NCOs for each channel while the TWO mode (c) uses both, although it reaches half of the modulation bandwidth. Finally, the HALF mode (d) combines one DUC from each channel pair to double the modulation bandwidth for half the channels.

peak of the combined envelope waveform and make sure that the worst case never goes beyond the clipping level. If the overall DAC range is normalized to the -1.0/+1.0 range, if any of the resulting RF the 0.5/+0.5 limits, then the resulting samples can be multiplied by two and then activate the numerical 6dB attenuator for that IQ pair so this will optimize signal quality without modifying the relative power of both signals.

Phase can also be set for each NCO. This is the initial phase for the NCO when operation starts. As the starting instant for all the NCOs in pair, module, or system is deterministic, the phase control allows for relative phase adjustment of all the carriers. This is especially meaningful when the carrier frequency is the same for all the channels. As all the NCOs are referred to the sampling clock, and this can be based in the same frequency and time references. The initial phase will be kept indefinitely, easing applications where relative phase control is mandatory, such as Phase-Array Radars, MIMO, Beamforming, or Quantum Computing. Phase, like frequency, can be changed "on the fly". This means that a new relative phase setting can be set (i.e. to change the direction of a beam) without interrupting signal generation, unlike some other DUC-equipped AWGs in the market.

The Proteus DUC can also be used in the NCO mode. In this mode, no IQ data is read, and the only working elements in the DUCs are the NCOs. This mode can be used to generate multiple CW signals without the need to define a "dummy" DC IQ waveform. When set to the same frequency, all the NCOs are coherent, and the relative phase can be controlled accurately, making this multi-channel CW RF generator highly suited for applications as Phase Array Radar, Beamforming, or any application where multiple L.O. with tightly controlled relative phases.

IQ Waveform Data Formats in Proteus

Once calculated, normalized, and scaled, IQ complex waveforms must be quantized and concerted to 16-bit unsigned integers, before being transferred to the target waveform memory. As previously mentioned, each channel pair (two of them in a single PXI module) shares the same DDR bank with a capacity of up to 16GSamples (8-bit mode) or 8GSamples (16-bit mode). Each bank can be segmented in up to 64K segments. Waveform segments are the real target for waveforms being downloaded. Real waveforms are stored as a series of samples read sequentially ("true arb" architecture). However, complex (IQ waveforms) cannot always be handled as two independent segments, as each channel can only access one segment at a time. The solution for this issue is reading complex waveforms as a single entity, so both components are stored in the same segment. The simplest way to do it, and the best one to minimize intermediate buffering, is storing IQ waveforms as interleaved pairs (I1, Q1, I2, Q2…, In, Qn). This is the format used for the ONE DUC mode in Proteus (fig. 4.3). This is the formatting procedure to follow:

1. Calculate I and Q waveforms
2. Joint normalization
3. Interleaving (I, Q, I, Q)
4. Download to target segment
5. Segment size = 2 x I/Q waveform size
6. Select segment for target channel (1-4)

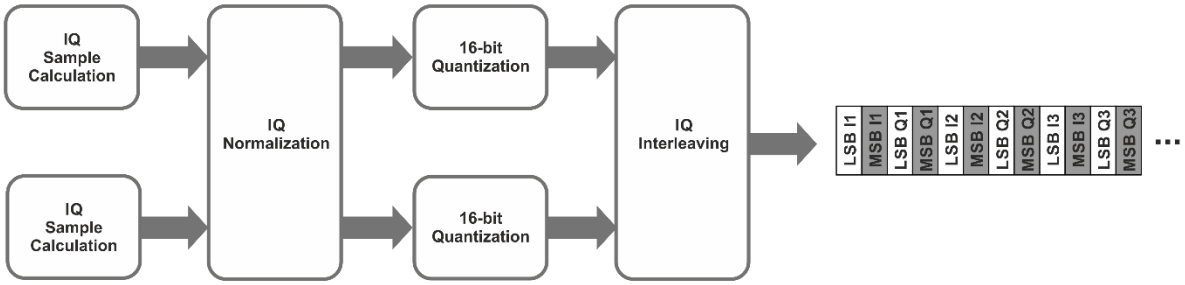


Figure 4.3: IQ waveform data must be stored properly for play-back. In the ONE mode, data must be arranged as a single segment with the I and Q samples interleaved.

The maximum overall data rate for this mode is 2.5GS/s (16-bit) so the maximum sample rate for each component would be 1.25GS/s (1.125GS/s for 9GS/s DAC conversion rate and 8x interpolation) and the resulting modulation BW would be slightly larger than 1.0 GHz.

The TWO mode is more complex, as two sets of IQ pairs must be transferred to a given channel. The resulting two sets of IQ pairs must be doubled interleaved to be downloaded to a single segment (fig. 4.4). The binary data to be sent to the segment must be properly formatted, so the transfer to the waveform memory is aligned with the DUC block requirements. This is the sequence of formatting actions to be carried out:

1. Arrange the 16-bit samples in the I1, Q1, Q2, I2 sequence
2. Split all the 16-bit samples in two bytes
3. For each group of four samples, take the MSB bytes following the interleaving sequence shown above (I1M, Q1M, Q2M, I2M)
4. You must perform the same operation for the LSB bytes (I1L, Q1L, Q2L, I2L)
5. Obtain the final waveform data by interleaving the MSB and LSB groups built in the previous steps (I1M, I1L, Q1M, Q1L, Q2M, Q2L, I2M, I2L)

As the overall data rate (5GBytes/s) stands here as well, the maximum sampling rate for each one of the IQ components is 625MS/s and modulation BW will be close to 600MHz. However, as interpolation factor depends on the ratio between the DAC sampling rate and the baseband interpolation ratio, and currently the maximum interpolation factor implemented in Proteus is 8x, the maximum DAC sampling rate supporting the two mode is 625 * 8 = 5,000MS/s = 5GS/s. Future product improvements will allow for higher interpolation factors (16x) so the TWO mode will be feasible up to the maximum DAC sampling rate (9GS/s).

Finally, the HALF mode uses half of one of the DUCs in each channel of a given pair, using just one of the DACs after adding the output of each block. In this case, waveform data is stored as two independent segments in the same DDR bank and segment assignment is done as a direct real only waveform to each participating channel. This is the formatting procedure:

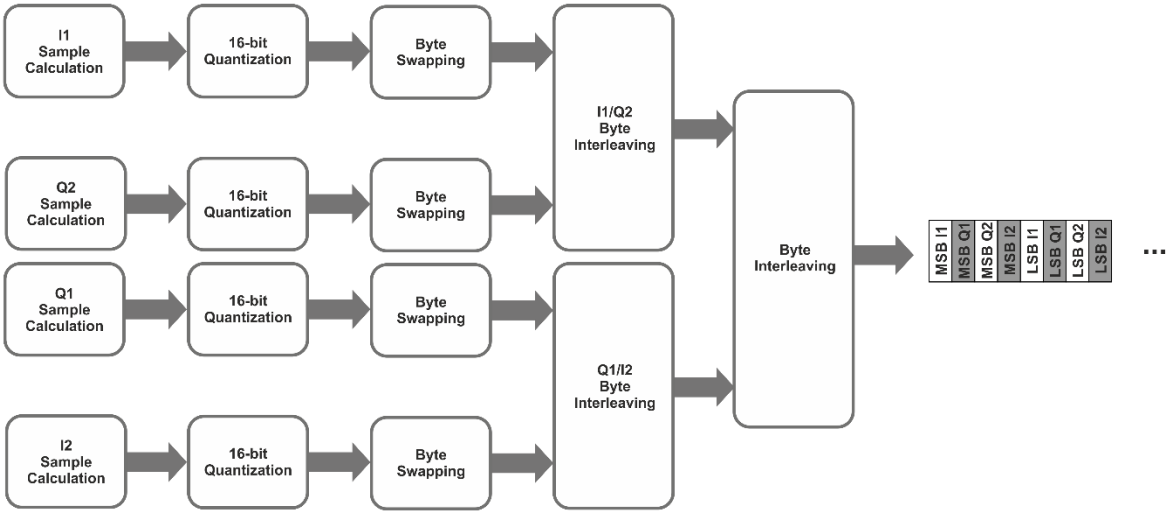


Figure 4.4: The TWO mode requires interleaving the IQ1 and IQ2 sample pairs together in such a way the DUC can use the data immediately and latency is minimized. Here, the dual-level interleaving process is shown.

1. Calculate I and Q waveforms
2. Joint normalization
3. Download I waveform to segment A
4. Download Q waveform to segment B
5. Segment size = I waveform size = Q waveform size
6. Select Segment A for target active channel (1 or 3)
7. Select Segment B for associated phantom channel (2 or 4)

NCO for each channel must be set to the same frequency and phase. When the mode is activated, the "phantom" channels will not output any signal, and the active channel will work exactly as it was in the ONE mode, although the I and Q quadrature modulated components come from different DUCs (each one using a different, but synchronized and coherent, NCO). The main advantage of this mode consists in increasing by a factor of 2 (up to 2.5GS/s) the sampling rate for each one of the components, so modulation BW goes beyond 2.3GHz. At a 9GS/s DAC sampling rate, and using the x4 interpolation factor, baseband sampling rate will be 2.25GS/s so modulation BW will go beyond 2GHz.

5 Appendix A – DUC Programming Example

The MATLAB script below calculates an IQ baseband signal and then uses the DUC to generate an RF modulated signal. It is possible to select the analog or digital modulation scheme and the parameters for it. It also allows for the selection of the Carrier Frequency.

```
% EXAMPLE FOR IQ MODE ONE IN PROTEUS USING VISA
%==============================================
% This example sets the IQ mode for the designated channel and downloads a
% complex(IQ) waveform to be applied to the built-in IQ modulator in the
% 'ONE' Mode. Analog and Digital Modulations are supported.

clc;
clear;

% Carrier Frequency
cfr = 1200E+06;
symbolRate = 150E6;
rollOff = 0.15;
samplingRate = 9E9; % change to 2.5E9 for P258X

% Set offset to any positive or negative frequency to shift carrier
fOffset = 0.0;
% Set initial phase for NCO
phase = 0.0 ;
%Set Target Channel
channel = 1;
%Set Target Segment
segment = 1;
% select reversed spectrum for generation in second Nyquist Zone
reverse = false;
% Boost Output Power by 6dB
apply6db = true;

fprintf(1, 'INITIALIZING SETTINGS\n');

% Communication Parameters
connStr = '192.168.1.48'; % your IP here
paranoia_level = 1; % 0, 1 or 2

%% Create Administrator
inst = TEProteusInst(connStr, paranoia_level);
fprintf('\n');

res = inst.Connect();
assert (res == true);

% If sampling rate lower than 2.5GHz, NCO frequency set to 500MHz
if samplingRate <= 2.5E+9
    cfr = 500E+6;
end


%Wfm Calculation
fprintf(1, 'Calculating WAVEFORM\n');

%ANALOG & DIGITAL MODULATION SETTINGS
% modType Analog:
% -1            Gaussian Pulse
% 0             AM
% 1             FM
% 2             PM
% 3             SSB;
% 4             CHIRP;
% modType Digital:
% 5             QPSK
% 6             QAM16
% 7             QAM32
% 8             QAM64
% 9             QAM128
%10             QAM256
%11             QAM512
%12             QAM1024
modType = 5;

% See CalculateAnalogModWfm Function to know the meaning of the param1 and
% param2 variables depending on the modulation scheme.
param1 = 100E-9; %90.0; %Peak Frequency Deviaton in Hz
param2 = 20E-9; %Modulaiton frequency in HZ
minCycles = 1; %Prime number is better
% Parameters for Digital Modulation
```

```
numOfSymbols = 16384;

% Interpolation according to DUC interpolation factor
interpol = 8; %8X interpolation factor
% Waveform granularity applies to the combined I/Q waveform so actual
% granularity for each component is granul / 2.
gCorr = 2;
intCorr = 1;

if modType <= 3
    % Calculate analog modulation I/Q waveforms
    wfmIq = CalculateAnalogModWfm(  modType,...
                                    minCycles,...
                                    samplingRate,...
                                    interpol / intCorr,...
                                    granul / gCorr,...
                                    param1,...
                                    param2);
else
    % Calculate QPSK/QAM I/Q Waveforms
    wfmIq = CalculateDigitalModWfm( modType,...
                                    numOfSymbols,...
                                    symbolRate,...
                                    rollOff,...
                                    samplingRate,...
                                    interpol / intCorr);
    % wfmIq length is not adjusted for granularity to optimize accuracy for
    % symbol rate. Howeverm symbol rate must be adjusted for signal loop
    % consistency. Actual Symbol Rate must be calculated and used in
    % analysis of the signal.
    actualSymbR = samplingRate / interpol * numOfSymbols / length(wfmIq);
    fprintf('\nActual Symbol Rate = to: %d\n', actualSymbR);
end

wfmIq = trimGran(wfmIq, granul / gCorr);

% I and Q waveforms
myWfmI = real(wfmIq);
myWfmQ = imag(wfmIq);
clear wfmIq;
% Negative Q waveform for inverse spectrum
if reverse
    myWfmQ = -myWfmQ;
end

% Frequency Offset applied
[myWfmI,  myWfmQ] = ApplyFreqOffset(    fOffset,...
                                        samplingRate / interpol,...
                                        myWfmI,...
                                        myWfmQ);

% I/Q data interleaving to a single array for downloadg
fprintf(1, 'I/Q INTERLEAVING\n');
% Envelope normalization to avoid DAC clipping
[myWfmI,  myWfmQ] = NormalIq(myWfmI, myWfmQ);
myWfm = Interleave(myWfmI, myWfmQ);
clear myWfmI myWfmQ;
% If necessary, wfm repetated for waveform granularity
myWfm = trimGran(myWfm, granul);

% SETTING AWG
fprintf(1, 'SETTING AWG\n');

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

% Set sampling rate for AWG to maximum.
inst.SendCmd([':FREQ:RAST ' num2str(2.5E9)]);
inst.SendCmd(sprintf(':INST:CHAN %d', channel));
% Interpolation factor for I/Q waveforms set to X8
inst.SendCmd(':SOUR:INT X8');
inst.SendCmd([':FREQ:RAST ' num2str(samplingRate)]);
% DAC Mode set to 'DUC' and IQ Modulation mode set to 'ONE'
inst.SendCmd(':MODE DUC');
inst.SendCmd(':IQM ONE');

% Waveform Downloading
% ********************
inst.SendCmd(':TRAC:DEL:ALL');
fprintf(1, 'DOWNLOADING WAVEFORM\n');
res = SendWfmToProteus(inst, channel, segment, myWfm, 16);
fprintf(1, 'WAVEFORM DOWNLOADED!\n');
clear myWfm;

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

```
% 6dB IQ Modulation gain applied
if apply6db
    inst.SendCmd(':NCO:SIXD2 ON');
else
    inst.SendCmd(':NCO:SIXD2 OFF');
end
% NCO frequency and phase setting
inst.SendCmd(sprintf(':NCO:CFR1 %d', cfr));
inst.SendCmd(sprintf(':NCO:PHAS1 %d', phase));
% Activate outpurt and start generation
inst.SendCmd(':OUTP ON');

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

% It is recommended to disconnect from instrument at the end
inst.Disconnect();
clear inst;
clear;
fprintf(1, 'END\n');

function finalWfm = trimGran(inWfm, granularity)
    % trimGran - Adjust wfm length for granularity
    %
    % Synopsis
    %   finalWfm = trimGran(inWfm, granularity)
    %
    % Description
    %   Repeat waveform the minmum number of times to meet the
    %   waveform length granularity criteria
    %
    % Inputs ([]s are optional)
    %   (double) inWfm  Input waveform
    %   (int16)  granularity
    %
    % Outputs ([]s are optional)
    %   (double) finalWfm Adjusted waveform

    baseL = length(inWfm);
    finaL = lcm(baseL, granularity);

    finalWfm = zeros(1, finaL);
    pointer = 1;

    while pointer < finaL
        finalWfm(pointer : (pointer + baseL -1)) = inWfm;
        pointer = pointer + baseL;
    end

end

function [rotI,  rotQ] = ApplyFreqOffset(fOffset, sampleRate, wfmI, wfmQ)

    wfmL = length(wfmI);
    fRes = sampleRate / wfmL;
    fOffset = round(fOffset / fRes) * fRes;

    cplexWfm = wfmI + 1i * wfmQ;
    clear wfmI wfmQ;
    angleArray = 0:(wfmL - 1);
    angleArray = 2 * pi * fOffset * angleArray;
    angleArray = angleArray / sampleRate;

    angleArray = exp(1i * angleArray);

    cplexWfm = cplexWfm .* angleArray;
    clear angleArray;

    rotI = real(cplexWfm);
    rotQ = imag(cplexWfm);
end

function [normI,  normQ] = NormalIq(wfmI, wfmQ)

    maxPwr = max(wfmI.*wfmI + wfmQ .* wfmQ);
    maxPwr = maxPwr ^ 0.5;

    normI = wfmI / maxPwr;
    normQ = wfmQ / maxPwr;

end

function outWfm = Interleave(wfmI, wfmQ)

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

```
        wfmLength =  length(wfmQ);
    end

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

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

function result = SendWfmToProteus( instHandle,...
                                    channel,...
                                    segment,...
                                    myWfm,...
                                    dacRes)

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

    % format Wfm
    myWfm = instHandle.Quantization(myWfm, dacRes);

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

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

    result = length(myWfm);
end

function waveform = CalculateAnalogModWfm(  modType,...
                                            minCycles,...
                                            sampleRate,...
                                            interpol,...
                                            granul,...
                                            param1,...
                                            param2)

    %ANALOG MODULATION WAVEFORM CALCULATION
    % modType = -1, GAUSSIAN; 0, AM; 1, FM; 2, PM; 3, SSB;

    %GAUSSIAN PULSE SETTINGS
    pulseLength = param1;
    pulseWidth = param2;


    %AM SETTINGS
    amModIndex = param1; %Modulation Index in %
    amModFreq = param2; %Modulation frequency in HZ

    %FM SETTINGS
    fmFreqDev = param1; %Peak Frequency Deviaton in Hz
    fmModFreq = param2; %Modulaition frequency in HZ

    %PM SETTINGS
    pmPhaseDev = param1; %Peak Phase Deviaton in Rads
    pmModFreq = param2; %Modulation frequency in HZ

    %SSB SETTINGS
    ssbModFreq = param2; %Modulation frequency in HZ

    %CHIRP SETTINGS
    chirpSweepRange = param1;
    chirpSweepTime = param2;

    %Waveform Length Calculation
    modFreq = amModFreq;

    if modType == -1
        modFreq = 1.0 / param1;
    elseif modType == 1
        modFreq = fmModFreq;
    elseif modType == 2
        modFreq = pmModFreq;
    elseif modType == 3
        modFreq = ssbModFreq;
    elseif modType == 4
        modFreq = 1 / chirpSweepTime;
    end
```

```
    actualSR = sampleRate / interpol;
    if modType ~= 4
        numOfSamples = round(actualSR / abs(modFreq / minCycles));
    else
        numOfSamples = round(actualSR / abs(modFreq));
    end
    totalNumOfSamples = numOfSamples;

    % As samples sent to the instrument are twice the number of complex
    % samples, granul must be defined as half the actual number

    numOfReps = 1;

    while modType ~= 4 && mod(totalNumOfSamples, granul) ~= 0
        totalNumOfSamples = totalNumOfSamples + numOfSamples;
        numOfReps = numOfReps + 1;
    end

    numOfSamples = totalNumOfSamples;
    fRes = actualSR / numOfSamples;

    % Round modFreq to the nearest integer number of Cycles

    modFreq = round(modFreq / fRes) * fRes;

    %Waveform calculation
    fprintf(1, 'WAVEFORM CALCULATION\n');

    waveform = 0: (numOfSamples - 1);
    waveform = (1 / actualSR) .* waveform;
    waveform = waveform - (numOfSamples / (2 * actualSR));

    if modType == -1
        sigma = pulseWidth /(2 * (2 * log(2)) ^0.5);%2.35;
        waveform = exp(-0.5 * (waveform/sigma).^2);
        waveform = waveform + 1i * waveform;
    elseif modType == 0
        waveform = 1 + amModIndex/100 .* sin(2 * pi * modFreq * waveform);
        waveform = waveform + 1i * waveform;
    elseif modType == 1
        fmFreqDev = round(fmFreqDev / fRes) * fRes;
        freqInst = fmFreqDev / modFreq * sin(2 * pi * modFreq * waveform);
        waveform = cos(freqInst) + 1i * sin(freqInst);
        clear freqInst;
    elseif modType == 2
        phaseInst = pmPhaseDev * sin(2 * pi * modFreq * waveform);
        waveform = cos(phaseInst) + 1i * sin(phaseInst);
        clear phaseInst;
    elseif modType == 3
        waveform = 2 * pi * modFreq * waveform;
        waveform = cos(waveform) + 1i * sin(waveform);
    elseif modType == 4
        chirpSweepRange = chirpSweepRange / 2;
        chirpSweepRange = round(chirpSweepRange / fRes) * fRes;
        freqInst = (actualSR * chirpSweepRange / numOfSamples) * waveform;
        freqInst = 2 * pi * freqInst .* waveform;
        waveform = cos(freqInst) + 1i * sin(freqInst);
        clear freqInst;
        waveform = trimGran(waveform, granul);
    end

    % waveform conditioning:
    waveform = waveform./((mean(abs(waveform).^2))^0.5);

end

function [dataOut] = CalculateDigitalModWfm(   modType,...
                                               numOfSymbols,...
                                               symbolRate,...
                                               rollOff,...
                                               sampleRate,...
                                               interpol)

    % modType     Modulation
    % 5             QPSK
    % 6             QAM16
    % 7             QAM32
    % 8             QAM64
    % 9             QAM128
    %10             QAM256
    %11             QAM512
    %12             QAM1024

    if modType == 5
        bitsPerSymbol = 2;
    elseif modType == 6
        bitsPerSymbol = 4;
    elseif modType == 7
        bitsPerSymbol = 5;
```

```
    elseif modType == 8
        bitsPerSymbol = 6;
    elseif modType == 9
        bitsPerSymbol = 7;
    elseif modType == 10
        bitsPerSymbol = 8;
    elseif modType == 11
        bitsPerSymbol = 9;
    elseif modType == 12
        bitsPerSymbol = 10;
    else
        bitsPerSymbol = 2;
    end

    % Waveform Length Calculation
    sampleRate = sampleRate / interpol;

    [decimation, oversampling] = reduceFraction(symbolRate, sampleRate);


    % Create IQ for QPSK/QAM
    % accuracy is the length of the shaping filter
    accuracy = 64;
    fType = 'sqrt'; % 'normal' or 'sqrt'
    % Get symbols in the range 1..2^bps-1
    data = getRnData(numOfSymbols, bitsPerSymbol);
    % Map symbols to I/Q constellation locations
    [dataI, dataQ] = getIqMap(data, bitsPerSymbol);
    % Adapt I/Q sample rate to the AWG's

    dataI = expanData(dataI, oversampling);
    dataQ = expanData(dataQ, oversampling);
    % Calculate baseband shaping filter
    rsFilter = rcosdesign(rollOff,accuracy,oversampling, fType);
    % Apply filter through circular convolution
    dataI = cconv(dataI, rsFilter, length(dataI));
    dataQ = cconv(dataQ, rsFilter, length(dataQ));

    dataI = dataI(1:decimation:length(dataI));
    dataQ = dataQ(1:decimation:length(dataQ));
    % Output waveforfm must be made of complex samples
    dataOut = dataI + 1i * dataQ;
end

function dataOut = getRnData(nOfS, bPerS)

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

function [symbI, symbQ] = 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;
```

```
end

function [outNum, outDen] = reduceFraction(num, den)
%reduceFraction Reduce num/den fraction
%   Use integers although not mandatory
    num = round(num);
    den = round(den);
    % Reduction is obtained by calcultaing the greater common divider...
    G = gcd(num, den);
    % ... and then dividing num and den by it.
    outNum = num / G;
    outDen = den / G;
end


function dataOut = expanData(inputWfm, oversampling)

    dataOut = zeros(1, oversampling * length(inputWfm));
    dataOut(1:oversampling:length(dataOut)) = inputWfm;

end
```

6 Appendix B – Proteus Comm Library

This is a MATLAB function library required by the script in Appendix A.

```
% ========================================================================
% Copyright (C) 2016-2021 Tabor-Electronics Ltd <http://www.taborelec.com/>
%
% This program is free software: you can redistribute it and/or modify
% it under the terms of the GNU General Public License as published by
% the Free Software Foundation, either version 2 of the License, or
% (at your option) any later version.
%
% This program is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
% GNU General Public License for more details.
%
% You should have received a copy of the GNU General Public License
% along with this program.  If not, see <http://www.gnu.org/licenses/>.
% ========================================================================
% Author: Nadav Manos, Fractions by Joan Mercade
% Date: May 17, 2021
% Version: 2.0.1


classdef TEProteusInst < handle
    % TEProteusInst: NI-VISA based connection to Proteus Instrument.


    properties
        ParanoiaLevel = 1; % Paranoia level (0:low, 1:normal, 2:high)
    end

    properties (SetAccess=private)
        ConnStr = ''; % The Connection-String
        ViSessn = 0;  % VISA Session
    end

    properties (Constant=true)
        VISA_IN_BUFF_SIZE = 8192000;   % VISA Input-Buffer Size (bytes)
        VISA_IN_BUFF_SIZE_LONG = 8192000;   % VISA Input-Buffer Size for Long Transfers (bytes)
        VISA_OUT_BUFF_SIZE = 8192000;  % VISA Output-Buffer Size (bytes)
        VISA_OUT_BUFF_SIZE_LONG = 8192000;  % VISA Output-Buffer Size for Long Transfers (bytes)
        VISA_TIMEOUT_SECONDS = 10;  % VISA Timeout (seconds)
        BINARY_CHUNK_SIZE = 409600;   % Binary-Data Write Chunk Size (samples)
        WAIT_PAUSE_SEC = 0.02;      % Waiting pause (seconds)
    end

    methods % public

        function obj = TEProteusInst(connStr, paranoiaLevel)
            % TEProteusInst - Handle Class Constructor
            %
            % Synopsis
            %   obj = TEProteusInst(connStr, [verifyLevel])
            %
            % Description
            %   This is the constructor of the VisaConn (handle) class.
            %
            % Inputs ([]s are optional)
            %   (string) connStr      connection string: either a full
            %                         VISA resource name, or an IP-Address.
            %   (int) [paranoiaLevel = 1] paranoia level [0,1 or 2].
            %
            % Outputs
            %   (class) obj      VisaConn class (handle) object.
            %

            assert(nargin == 1 || nargin == 2);

            ipv4 = '^(?:[0-9]{1,3}\.){3}[0-9]{1,3}$';
            if 1 == regexp(connStr, ipv4)
```

```
            connStr = sprintf('TCPIP0::%s::5025::SOCKET', connStr);
        end

        if nargin == 2
            %verifyLevel = varargin(1);
            if paranoiaLevel < 1
                obj.ParanoiaLevel = 0;
            elseif paranoiaLevel > 2
                obj.ParanoiaLevel = 2;
            else
                obj.ParanoiaLevel = fix(paranoiaLevel);
            end
        else
            obj.ParanoiaLevel = 1;
        end

        obj.ConnStr = connStr;
        % Select the right one for the active VISA Library
        obj.ViSessn = visa('NI', connStr);
        %obj.ViSessn = visa('keysight', connStr);
        %obj.ViSessn = visa('tek', connStr);

        set(obj.ViSessn, 'OutputBufferSize', obj.VISA_OUT_BUFF_SIZE);
        set(obj.ViSessn, 'InputBufferSize', obj.VISA_IN_BUFF_SIZE);
        obj.ViSessn.Timeout = obj.VISA_TIMEOUT_SECONDS;
        %obj.ViSessn.Terminator = newline;

    end

    function delete(obj)
        % delete - Handle Class Destructor
        %
        % Synopsis
        %   obj.delete()
        %
        % Description
        %   This is the destructor of the VisaConn (handle) class.
        %   (to be called on a VisaConn class object).
        %

        obj.Disconnect();
        delete(obj.ViSessn);
        obj.ViSessn = 0;
    end

    function ok = Connect(obj)
        % Connect - open connection to remote instrument.
        %
        % Synopsis
        %    ok = obj.Connect()
        %
        % Description
        %    Open connection to the remote instrument
        %
        % Outputs
        %    (boolean) ok   true if succeeded; otherwise false.
        %

        ok = false;
        try
            if strcmp(obj.ViSessn.Status, 'open')
                ok = true;
            else
                fopen(obj.ViSessn);
                pause(obj.WAIT_PAUSE_SEC);
                ok = strcmp(obj.ViSessn.Status, 'open');
            end
        catch ex
            msgString = getReport(ex);
            warning('fopen failed:\n%s',msgString);
        end
    end

    function Disconnect(obj)
        % Disconnect - close connection to remote instrument.
        %
        % Synopsis
        %   obj.Disconnect()
```

```
    %
    % Description
    %    Close connection to remote-instrument (if open).

    if strcmp(obj.ViSessn.Status, 'open')
        stopasync(obj.ViSessn);
        flushinput(obj.ViSessn);
        flushoutput(obj.ViSessn);
        fclose(obj.ViSessn);
    end
end

function [errNb, errDesc] = QuerySysErr(obj, bSendCls)
    % QuerySysErr - Query System Error from the remote instrument
    %
    % Synopsis
    %   [errNb, [errDesc]] = obj.QuerySysErr([bSendCls])
    %
    % Description
    %   Query the last system error from the remote instrument,
    %   And optionally clear the instrument's errors list.
    %
    % Inputs ([]s are optional)
    %   (bool) [bSendCls = false]
    %           should clear the instrument's errors-list?
    %
    % Outputs ([]s are optional)
    %   (scalar) errNb     error number (zero for no error).
    %   (string) [errDesc] error description.

    if ~exist('bSendCls', 'var')
        bSendCls = false;
    end

    obj.waitTransferComplete();
    [answer, count, errmsg] = query(obj.ViSessn, 'SYST:ERR?');
    obj.waitTransferComplete();

    if ~isempty(errmsg)
        error('getError() failed: %s', errmsg);
    end

    sep = find(answer == ',');
    if (isempty(sep) || count <= 0 || answer(count) ~= newline)
        warning('querySysErr() received invalid answer: "%s"', answer);
        flushinput(obj.ViSessn);
    end

    if ~isempty(sep) && isempty(errmsg)
        errNb = str2double(answer(1:sep(1) - 1));
        errmsg = answer(sep(1):end);
        if 0 ~= errNb && nargin > 1 && bSendCls
            query(obj.ViSessn, '*CLS; *OPC?');
        end
    else
        errNb =  -1;
        if isempty(errmsg)
            errmsg = answer;
        end
    end

    if nargout > 1
        errDesc = errmsg;
    end
end


function SendCmd(obj, cmdFmt, varargin)
    % SendCmd - Send SCPI Command to instrument
    %
    % Synopsis
    %   obj.SendCmd(cmdFmt, ...)
    %
    % Description
    %   Send SCPI Command to the remote instrument.
    %
    % Inputs ([]s are optional)
```

```
        %   (string) cmdFmt      command string-format (a la printf).
        %            varargin    arguments for cmdFmt
        obj.waitTransferComplete();

        if nargin > 2
            cmdFmt = sprintf(cmdFmt, varargin{1:end});
        end

        resp = '';
        errMsg = '';
        respLen = 0;

        if obj.ParanoiaLevel == 0
            fprintf(obj.ViSessn, cmdFmt);
            obj.waitTransferComplete();
        elseif obj.ParanoiaLevel == 1
            cmdFmt = strcat(cmdFmt, ';*OPC?');
            [resp, respLen, errMsg] = query(obj.ViSessn, cmdFmt);
        elseif obj.ParanoiaLevel >= 2
            cmdFmt = strcat(cmdFmt, ';:SYST:ERR?');
            [resp, respLen, errMsg] = query(obj.ViSessn, cmdFmt);
        end

        if (obj.ParanoiaLevel > 0 && ~isempty(errMsg))
            error('query(''%s\'') failed\n %s', cmdFmt, errMsg);
        elseif (obj.ParanoiaLevel >= 2 && respLen > 0)
            resp = deblank(resp);
            sep = find(resp == ',');
            if ~isempty(sep)
                errNb = str2double(resp(1:sep(1) - 1));
                if 0 ~= errNb
                    query(obj.ViSessn, '*CLS; *OPC?');
                    warning('System Error #%d after ''%s'' (%s).', ...
                        errNb, cmdFmt, resp);
                end
            end
        end
    end

    function resp = SendQuery(obj, qformat, varargin)
        % SendQuery - Send SCPI Query to instrument
        %
        % Synopsis
        %   resp = obj.SendQuery(qformat, ...)
        %
        % Description
        %   Send SCPI Query to the remote instrument,
        %   And return the instrument's response (string).
        %
        % Inputs ([]s are optional)
        %   (string) qformat     query string-format (a la printf).
        %            varargin    arguments for qformat
        %
        % Outputs ([]s are optional)
        %   (string) resp     the instrument's response.

        obj.waitTransferComplete();
        if nargin == 2
            [resp, respLen, errMsg] = query(obj.ViSessn, qformat);
        elseif nargin > 2
            qformat = sprintf(qformat, varargin{1:end});
            [resp, respLen, errMsg] = query(obj.ViSessn, qformat);
        else
            resp = '';
            errMsg = '';
            respLen = 0;
        end

        if ~isempty(errMsg)
            error('query(''%s\'') failed\n %s', qformat, errMsg);
        end

        if respLen > 0
            % remove trailing blanks
            resp = deblank(resp);
        end
    end
```

```
function SendBinaryData(obj, pref, datArray, elemType)
    % SendBinaryData - Send binary data to instrument
    %
    % Synopsis
    %   obj.SendBinaryData(pref, datArray, elemType)
    %
    % Description
    %   Send array of basic-type elements to the remote instrument
    %   as binary-data with binary-data header and (optional) SCPI
    %   statement prefix (e.g. ":TRAC:DATA").
    %
    % Inputs ([]s are optional)
    %   (string) pref      SCPI statement (e.g. ":TRAC:DATA")
    %                      sent before the binary-data header.
    %   (array)  datArray  array of fixed-size elements.
    %   (string) elemType  element type name (e.g. 'uint8')

    obj.waitTransferComplete();



    if ~exist('pref', 'var')
        pref = '';
    end
    if ~exist('datArray', 'var')
        datArray = [];
    end
    if ~exist('elemType', 'var')
        elemType = 'uint8';
        datArray = typecast(datArray, 'uint8');
    end

    numItems = length(datArray);
    switch elemType
        case { 'int8', 'uint8' 'char' }
            itemSz = 1;
        case { 'int16', 'uint16' }
            itemSz = 2;
        case { 'int32', 'uint32', 'single' }
            itemSz = 4;
        case { 'int64', 'uint64', 'double' }
            itemSz = 8;
        otherwise
            error('unsopported element-type ''%s''', elemType);
    end

    assert(itemSz >= 1 && itemSz <= obj.BINARY_CHUNK_SIZE);

    getChunk = @(offs, len) datArray(offs + 1 : offs + len);

    % make binary-data header
    szStr = sprintf('%lu', numItems * itemSz);
    pref = sprintf('*OPC?;%s#%u%s', pref, length(szStr), szStr);
    % send it (without terminating new-line!):
    fwrite(obj.ViSessn, pref, 'char');
    obj.waitTransferComplete();

    % send the binary-data (in chunks):
    offset = 0;
    chunkLen = fix(obj.BINARY_CHUNK_SIZE / itemSz);
    while offset < numItems
        if offset + chunkLen > numItems
            chunkLen = numItems - offset;
        end
        dat = getChunk(offset, chunkLen);
        fwrite(obj.ViSessn, dat, elemType);
        obj.waitTransferComplete();
        offset = offset + chunkLen;
    end

    % read back the response to that *OPC? query:
    q = fscanf(obj.ViSessn, '%s');
    %fgets(obj.ViSessn, 2);

    if obj.ParanoiaLevel >= 2
        [errNb, errDesc] = obj.QuerySysErr(1);
        if 0 ~= errNb
```

```
            warning('System Error #%d (%s) after sending ''%s ..''.', errNb, errDesc, pref);
        end
    end
end

function datArray = ReadBinaryData(obj, pref, elemType)
    % ReadBinaryData - Read binary data from instrument
    %
    % Synopsis
    %   datArray = obj.ReadBinaryData(pref, elemType)
    %
    % Description
    %   Read array of basic-type elements from the instrument.
    %
    % Inputs ([]s are optional)
    %   (string) pref      SCPI statement (e.g. ":TRAC:DATA")
    %                      sent before the binary-data header.
    %   (string) elemType  element type name (e.g. 'uint8')
    %
    % Outputs ([]s are optional)
    %   (array)  datArray  array of fixed-size elements.

    obj.waitTransferComplete();

    %set(obj.ViSessn, 'InputBufferSize', obj.VISA_IN_BUFF_SIZE_LONG);

    if ~exist('pref', 'var')
        pref = '';
    end

    switch elemType
        case { 'int8', 'uint8' 'char' }
            itemSz = 1;
        case { 'int16', 'uint16' }
            itemSz = 2;
        case { 'int32', 'uint32', 'single' }
            itemSz = 4;
        case { 'int64', 'uint64', 'double' }
            itemSz = 8;
        otherwise
            error('unsopported element-type ''%s''', elemType);
    end

    assert(itemSz >= 1 && itemSz <= obj.BINARY_CHUNK_SIZE);

    % Send the prefix (if it is not empty)
    if ~isempty(pref)
        fprintf(obj.ViSessn, pref);
    end
    obj.waitTransferComplete();

    % Read binary header
    while true
        ch = fread(obj.ViSessn, 1, 'char');
        if ch == '#'
            break
        end
    end

    % Read the first digit
    ch = fread(obj.ViSessn, 1, 'char');
    assert ('0' < ch && ch <= '9');

    ndigits = ch - '0';
    %fprintf('ReadBinaryData: ndigits = %d\n', ndigits);

    sizestr = fread(obj.ViSessn, ndigits, 'char');
    numbytes = 0;
    for n = 1:ndigits
        ch = sizestr(n, 1);
        numbytes = numbytes * 10 + (ch - '0');
    end

    %fprintf('ReadBinaryData: numbytes = %d\n', numbytes);

    datLen = ceil(numbytes / itemSz);
    assert(datLen * itemSz == numbytes);
```

```
    datArray = zeros(1, datLen, elemType);

    chunkLen = fix(obj.BINARY_CHUNK_SIZE / itemSz);

    %fprintf('ReadBinaryData: datLen=%d, chunkLen=%d\n', datLen, chunkLen);

    % send the binary-data (in chunks):
    offset = 0;

    while offset < datLen
        if datLen - offset < chunkLen
            chunkLen = datLen - offset;
        end
        datArray(offset + 1 : offset + chunkLen) = ...
            fread(obj.ViSessn, chunkLen, elemType);
        %obj.waitTransferComplete();
        offset = offset + chunkLen;
    end

    % read the terminating newline character
    ch = fread(obj.ViSessn, 1, 'char');
    assert(ch == newline);

    set(obj.ViSessn, 'InputBufferSize', obj.VISA_IN_BUFF_SIZE);
end

function model = identifyModel(obj)
    idnStr = obj.SendQuery('*IDN?');
    idnStr = split(idnStr, ',');

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

    model = char(model);
end

function options = getOptions(obj)
    optStr = obj.SendQuery('*OPT?');
    options = split(optStr, ',');
end

function maxSr = getMaxSamplingRate2(obj, model)

    maxSr = 9.0E+9;

    if contains(model, 'P258')
        maxSr = 2.5E+9;
    elseif contains(model, 'P128')
        maxSr = 1.25E+9;
    end
end

function maxSr = getMaxSamplingRate(obj)
    maxSr = obj.SendQuery(':FREQ:RAST MAX?');
    maxSr = str2double(maxSr);
end

function minSr = getMinSamplingRate2(obj, model)

    minSr = 1.0E+9;
end

function minSr = getMinSamplingRate(obj)
    minSr = obj.SendQuery(':FREQ:RAST MIN?');
    minSr = str2double(minSr);
end

function granularity = getGranularity(obj, model, options)

    flagLowGranularity = false;

    for i = 1:length(options)
```

```
        if contains(options(i), 'LWG')
            flagLowGranularity = true;
        end
    end

    sR = obj.SendQuery(':FREQ:RAST?');
    sR = str2double(sR);
    % For P9082 and P9484 granularity is 64 for SR > 2.5E9
    granularity = 64;
    if flagLowGranularity && sR<=2.5E9
        granularity = 32;
    end

    if contains(model, 'P258')
        granularity = 32;
        if flagLowGranularity
            granularity = 16;
        end
    elseif contains(model, 'P128')
        granularity = 32;
        if flagLowGranularity
            granularity = 16;
        end
    end
end

function numOfChannels = getNumOfChannels(obj, model)

    numOfChannels = 4;

    if contains(model, 'P9082')
        numOfChannels = 2;
    elseif contains(model, 'P9482')
        numOfChannels = 2;
    elseif contains(model, 'P1282')
        numOfChannels = 2;
    elseif contains(model, 'P2582')
        numOfChannels = 2;
    end
end

function dacRes = getDacResolution2(obj, model)

    dacRes = 16;

    if contains(model, 'P908')
        dacRes = 8;
    end
end

function dacRes = getDacResolution(obj)

    dacRes = obj.SendQuery(':TRAC:FORM?');

    if contains(dacRes, 'U8')
        dacRes = 8;
    else
        dacRes = 16;
    end
end

function retval = Quantization (obj, myArray, dacRes)

  minLevel = 0;
  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
```

```
    end % public methods

    methods (Access = private) % private methods

        function waitTransferComplete(obj)
            % waitTransferComplete - wait till transfer status is 'idle'
            while ~strcmp(obj.ViSessn.TransferStatus,'idle')
                pause(obj.WAIT_PAUSE_SEC);
            end
        end
    end % private methods

end
```

Document Revision History

Table Document Revision History

Revision	Date	Description	Author
1.0	26-October-21	• Original release.	Joan Mercade joan@taborelec.com

Acronyms & Abbreviations

Table Acronyms & Abbreviations

Acronym	Description
µs or us	Microseconds
ACPR	Adjacent Channel Power Ratio
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
CCDF	Complementary Cumulative Distribution Function
CW	Continuous Wave
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

Acronym	Description
DNL	Differential Non-Linearity
DSO	Digital Storage Oscilloscope
DUC	Digital Up-Converter
DUT	Device Under Test
ENoB	Effective Number of Bits
ESD	Electrostatic Discharge
EVM	Error Vector Magnitude
FPGA	Field-Programmable Gate Arrays
FW	Firmware
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
IMD	Intermodulation Distortion
INL	Integral Non-Linearity
I/O	Input / Output
IP	Internet Protocol
IQ	In-phase Quadrature
IVI	Interchangeable Virtual Instrument
JSON	JavaScript Object Notation
kHz	Kilohertz
LCD	Liquid Crystal Display
LO	Local Oscillator
MAC	Media Access Control (address)
MDR	Mini D Ribbon (connector)

Acronym	Description
MHz	Megahertz
ms	Milliseconds
NCO	Numerically Controlled Oscillator
ns	Nanoseconds
OFDM	Orthogonal Frequency-Division Multiplexing
PAM	Pulse-amplitude Modulation
PAPR	Peak-to-Average Power Ratio
PC	Personal Computer
PCAP	Projected Capacitive Touch Panel
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PXI	PCI eXtension for Instrumentation
PXIe	PCI Express eXtension for Instrumentation
QC	Quantum Computing
Qubits	Quantum bits
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
SINAD	Signal-to-Noise-And-Distortion Ratio
SMA	Subminiature version A connector
SMP	Subminiature Push-on connector
SNR	Signal-to-Noise Ratio
SPI	Serial Peripheral Interface

Acronym	Description
SQNR	Signal to Quantization Noise Signal
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

Resources & Contact

For more information on Microwave signal generation challenges and solutions, review the following resources:

- White Paper: Multi-Nyquist Zones Operation-Solution Note
- White Paper: Direct Generation/Acquisition of Microwave Signals
- White Paper: Effective Number of Bits for Arbitrary Waveform Generators
- White Paper: Multi-Tone Signal Generation with AWGs
- Solution Brief**:** Envelope Tracking – Solution Note
- **Download Data Sheet**

Stay Up to Date

- www.taborelec.com
- LinkedIn page
- YouTube channel

Corporate Headquarters

Address: 9 Hata'asia St., 3688809 Nesher, Israel
Phone: (972) 4 821 3393
Fax: (972) 4 821 3388

For Information

Email: info@tabor.co.il

For Service & Support

Email: support@tabor.co.il

US Sales & Support (Astronics)

Address: 4 Goodyear Irvine, CA 92618
Phone: (800) 722 2528
Fax: (949) 859 7139

For Information

Email: info@taborelec.com

For Service & Support

Email: support@taborelec.com