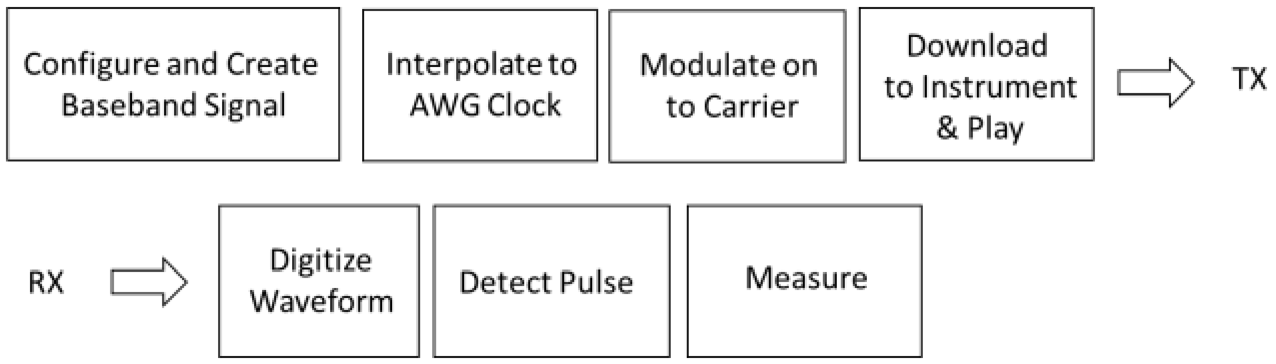
Generating and Measuring Communications Signals with the Proteus AWT

Introduction

Creating and analyzing signals with Proteus and MATLAB takes a few simple steps. In this application note we show how to generate and receive a WLAN beacon signal at 2.4GHz in the instruments' first Nyquist Zone. The code can easily be modified to create a signal in the second Nyquist zone, all the way up to the WiFi-6 frequency extension of 7.125GHz.



Configure and Create the Baseband Signal

The beacon frame is a type of management frame. It identifies a basic service set (BSS) formed by a number of 802.11 devices. The access point periodically transmits the beacon frame to establish and maintain the network. The beacon frame consists of a MAC header, a beacon frame body and a valid frame check sequence (FCS). The beacon frame body contains the information fields which allows stations to associate with the network. A WLAN beacon frame is created using the **wlanMACFrame** function. We use the helper function **helperGenerateBeaconFrame** and we configure for non-high throughput operation. The beacon frame is encoded and modulated using the **wlanWaveformGenerator** function to create a baseband beacon packet.

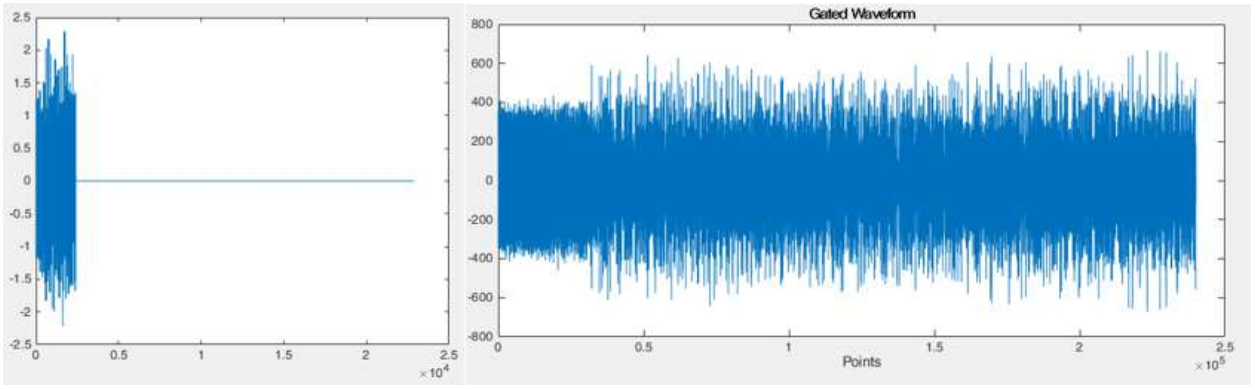
```
SSID = 'TABOR_AWG'; % Network SSID
beaconInterval = 1; % In Time units (TU)
band = 2.4;         % Band, 5 or 2.4 GHz
chNum = 3;          % Channel number, corresponds to 2422MHz
Fc = 2.422E+09;     % The center frequency of CH 3

% Generate Beacon frame
[mpduBits,fc] = helperGenerateBeaconFrame(chNum, band, beaconInterval, SSID);
cfgNonHT = wlanNonHTConfig;                % Create a wlanNonHTConfig object
cfgNonHT.PSDULength = numel(mpduBits)/8; % Set the PSDU length in bits
waveform = wlanWaveformGenerator(mpduBits, cfgNonHT, 'IdleTime',
beaconInterval*1024e-6);
```

5G Band operation

```
band = 5;           % Band, 5 or 2.4 GHz
chNum = 140;        % Channel number, corresponds to 5700MHz
Fc = 5.7E+09;
```

Plotting **waveform** yields the following result:



Interpolate to AWG Clock

The sample rate is 20MS/s, and this directly equates to 20MHz of signal Bandwidth. For first Nyquist operation we can set the instrument's sample rate to 9GS/s. In the example we define **Fs** as the sample rate of the baseband signal and **sclk** as the sample rate of the instruments sample clock.

```
%% re-sample to sclk of AWG
Fs = wlanSampleRate(cfgNonHT);            % Get the input sampling rate
sclk = 9e9;
FsNew = sclk/Fs;
waveformReSamp = IqIdealInterpolationWifi (waveform, FsNew);
```

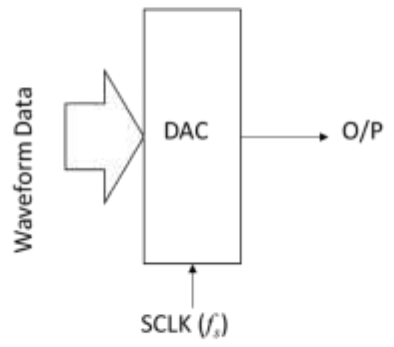
Modulate onto carrier

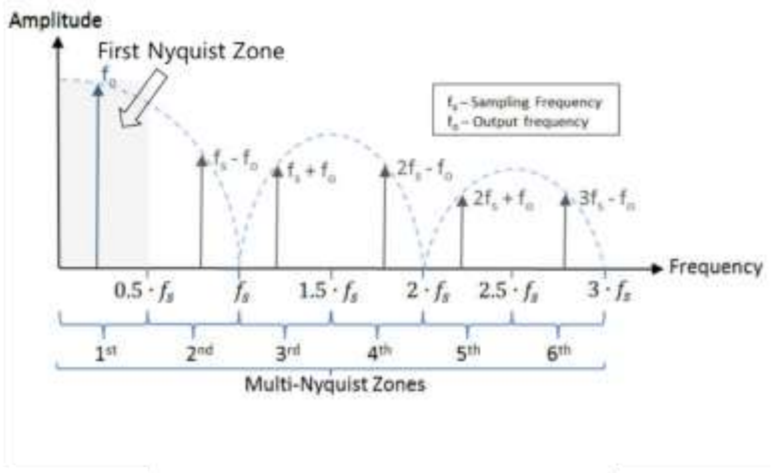
First, we create the carrier wave array;

```
 % Carrier Waveform creation
carrierWave = 0:(length(waveformReSamp) - 1);
carrierWave = carrierWave ./ sclk;
```

If we want to generate a signal in the 5G-band, we will generate the signal in the second Nyquist zone. Referring to the figure bellow - **Fc** would calculate as follows **Fc** = **sclk - Fc;** or 3.3GHz in the first Nyquist and 5.7GHz in the second Nyquist. A high pass filter could be used to attenuate the signal at 3.3GHz.

```
% second Nyquist band generation
if Fc > sclk / 2
    Fc = sclk - Fc;
    % one way to reverse the spectrum is changing the sign of the time so
    % carrier rotation in the complex plane goes in the opposite direction
    carrierWave = -carrierWave;
end
```





Finally, the following code creates the carrier and modulates the interpolated baseband signal on to the carrier **Fc**.

```
%% Modulate onto carrier
Fc = round(Fc / (sclk / length(carrierWave)) * sclk / length(carrierWave);
% Carrier generation
carrierWave = exp(1i * 2 * pi * Fc * carrierWave);
% Complex carrier multiplied by complex baseband waveform (IQ modulation)
%Modulated signal is just the real part of the complex product
waveformReSamp = real(waveformReSamp .* carrierWave);
waveformReSamp = waveformReSamp.';
```

Next, we format and scale the waveform in preparation for download;

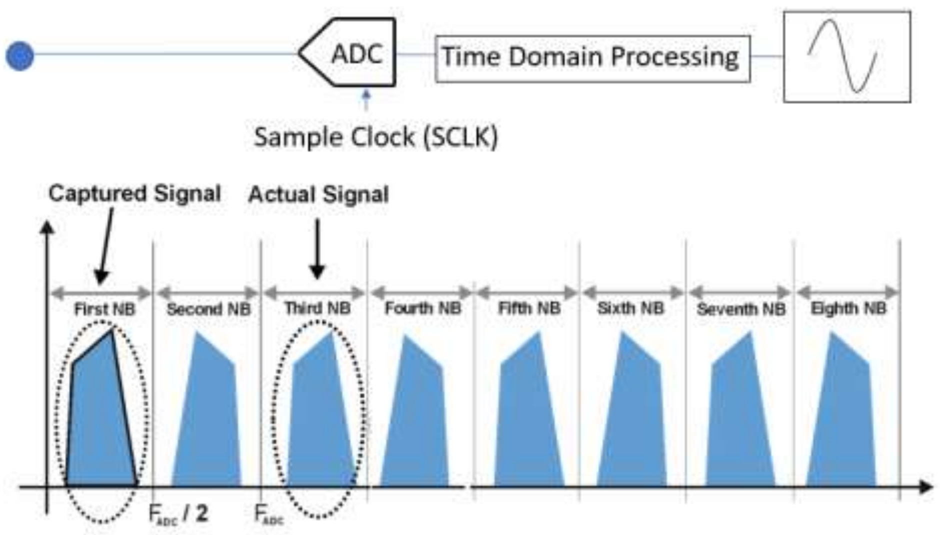
```
waveformReSampTrunk = waveformReSamp(1:10240000); %truncate - divisible by 64
bits=8;
dacSignal = ampScale(bits, waveformReSampTrunk);
```

The signal **`dacSignal`** is now ready to be downloaded to the instrument.

```
res = inst.SendScpi('*RST'); % Reset Instrument
res = inst.SendScpi(':FREQ:RAST 9E9'); % Set SLCK
res = inst.SendScpi('INST:CHAN 1'); % Enable channel 1
res = inst.SendScpi(':TRAC:DEF 1,10240000'); % Define a trace
res = inst.WriteBinaryData(':TRAC:DATA 0,#', dacSignal);
res = inst.SendScpi(':TRAC:SEL 1');
res = inst.SendScpi(':SOUR:FUNC:SEG 1');
res = inst.SendScpi(':OUTP ON');
```

Digitize the Waveform

We will digitize the waveform using the Nyquist Zone principles discussed earlier. We do this as the maximum sample clock frequency of the Proteus Digitizer is 5.4GS/s. This means the that 2.7GHz is the theoretical frequency limit within the first Nyquist Zone. Our Transmit signal is 2.442GHz, while it falls within the theoretical range, at 2.7GHz you would only get 2 sample points per period. 2.442GHz offers a few more sample points per period more. A more logical approach, and one that would yield in improved signal fidelity, would be to set the sample clock to 2GS/s.



Refering to the above figure – when we sample a 2.4GHz signal with a 2GS/s clock (F_{ADC} or SCLK) we will see an undersampled image of the signal in the first Nyquist Zone or at 400MHz (2.4GHz-2GS/s).

The following code sets the ADC up for an acquisition.

```
sampleRateADC = 2e9;
memoryAlloc = 10240000/4 % Just capture 25% of the waveform or use a trigger
readLen = 10240000/4;
readSize = uint64(readLen);
readOffset = uint64(0);
netArray = NET.createArray('System.UInt16', readLen);
rc = inst.AllocAdcReservedSpace(memoryAlloc);
rc = inst.SetAdcDualChanMode(1); % Turn on ADC dual-channels mode (state = 1)
rc = inst.SetAdcSamplingRate(sampleRateADC);
rc = inst.SetAdcCaptureSize(memoryAlloc);
rc = inst.SetAdcCaptureOffset(0);
```

The following code executes the acquisition and stores it in the memory we previously allocated.

```
status = inst.ReadAdcCaptureDoneStatus();
for i = 1 : 2500  % Wait till the capture completes

    if status ~= 0
        break;
    end
    status = inst.ReadAdcCaptureDoneStatus();
end
rc = inst.ReadAdcChanData(chanIndex, readSize, readOffset, netArray);
samples = int16(netArray);
```

This results in 2.5 million samples being stored in the int16 variable **samples**. With our sample clock set to 2GS/s this is 1.25ms of captured data.

Detecting the Pulse

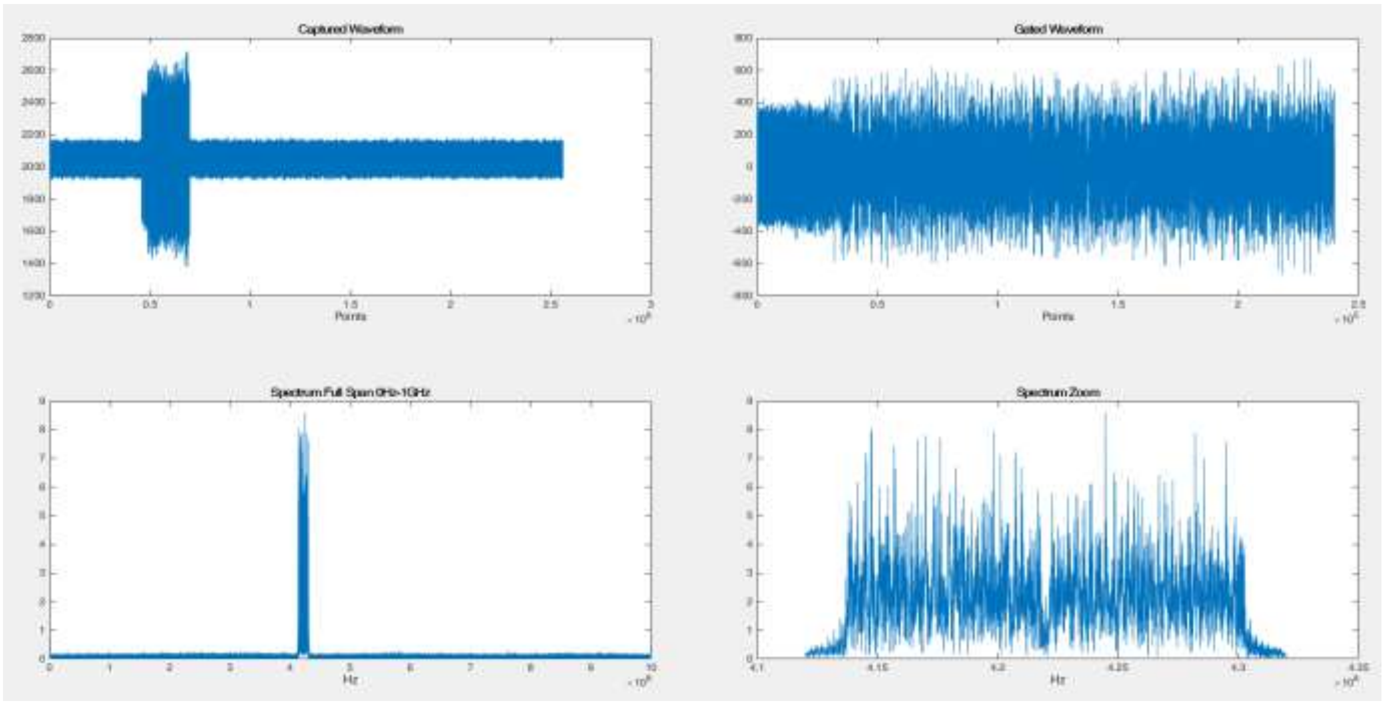
In the simple example of acquisition explained above, we randomly trigger the acquisition and set a capture time that has a high probability of capturing the pulse. Using one of the many trigger functions of Proteus we could use an external trigger and set a capture time that is equal to the pulse length, or trigger on the pulses own first rising edge and gain set the capture time to equal the pulse length.

Our capture is stored as a 16bit integer number. The peak values are < 2^{16} and the minimum values are just above zero so one of the first things we should do is normalize **samples** so the waveform is oscillating around zero. An easy quick way to do this is to take the mean value of **samples** and subtract it, effectively reducing the inherent DC level and preparing it for the measurement.

```
meanSig = mean(samples);
dataReadTimeDC=samples-meanSig;
```

Making a Measurement

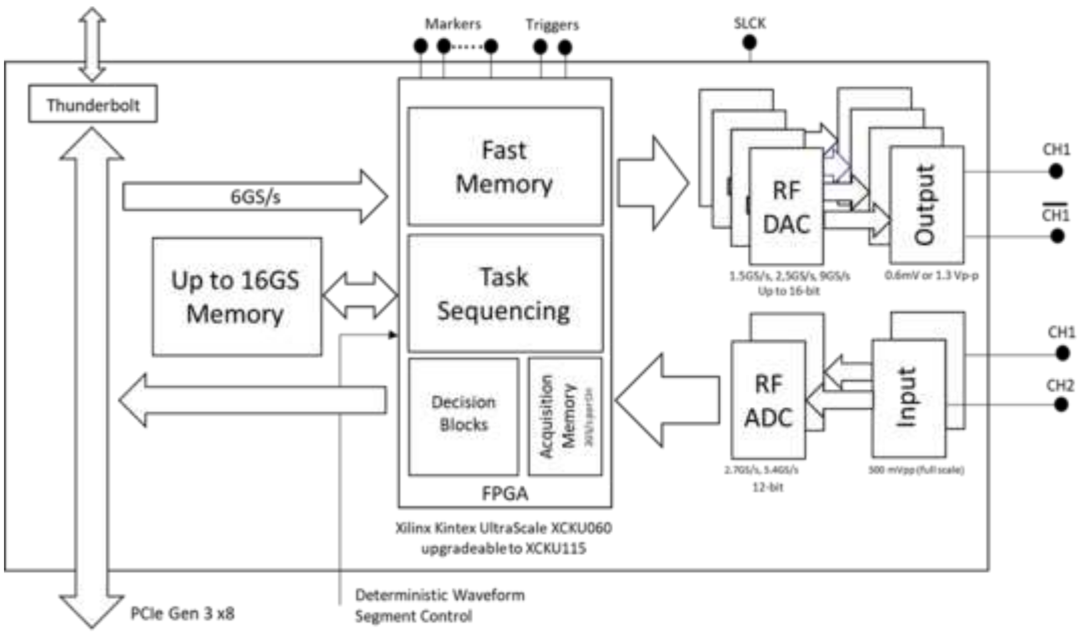
Now the signal is ready to perform measurements. In the following example I've created 4 plots. The first upper left quadrant is the time domain display of the base acquisition, then in the second upper quadrant using a simple software trigger I capture only the pulse itself. The lower plots are broad spectrum 0-1GHz and tuned spectrum with a center frequency of 442MHz.



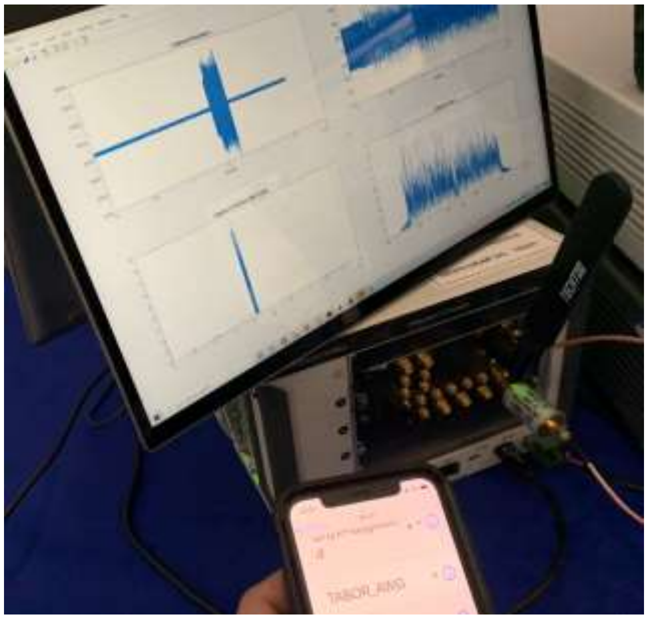
The above plots also have random noise added and I used a filter to eliminate any spurious signals. More measurements are possible using the rich signal processing toolset within MATLAB including modulation quality, adjacent channel power and CCDF.

Conclusion

The unique architecture of an AWT allows for wideband signal generation and analysis. The Proteus system uses the latest ADC's and DAC's combined with a powerful FPGA that we can also utilize for further signal processing.



Finally, just for fun we can put an antenna on the differential output of the AWT and see if we can pick-up the beacon signal!



The code examples can be found on our Github site – please provide us with your Github name and we will authorize you as a collaborator.