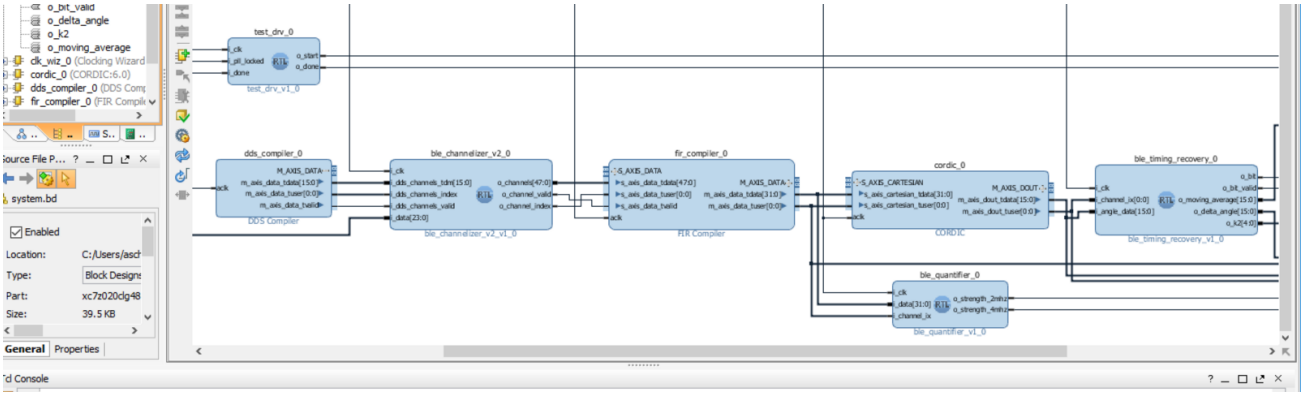
Andreas Schweizer
Oct 24, 2017 · 12 min read

On Developing a Wideband BLE Sniffer—Part 2



In the first part, I've explained the "what" and "why" of this endeavor. Progress was slow during the past few months, not only because we had many other things going on, but also because the Vivado project had grown to a size where it took about 40 minutes from the start of a synthesis run to completing the bitstream generation for the FPGA.

A few weeks ago, I decided to change the approach. Instead of working on the full implementation, I split the project into smaller pieces. First, I recorded the output of the AD9364 for a couple of BLE advertisements into a memory buffer and exported it via the Zynq UART into a text file. Second, I created a new project in Vivado which only contains the parts that process the output from the AD9364 analog-to-digital converter to generate the bitstream of the BLE packets. Let's call this our *baseband processor*. And third, I set up a testbench which reads the recorded RF samples, feeds them into the baseband processor and stores the result back into a text file. The synthesis of the baseband processor, including post-synthesis simulation, takes less than 5 minutes, which is much more fun for development.

So, let's see how the baseband processor works.

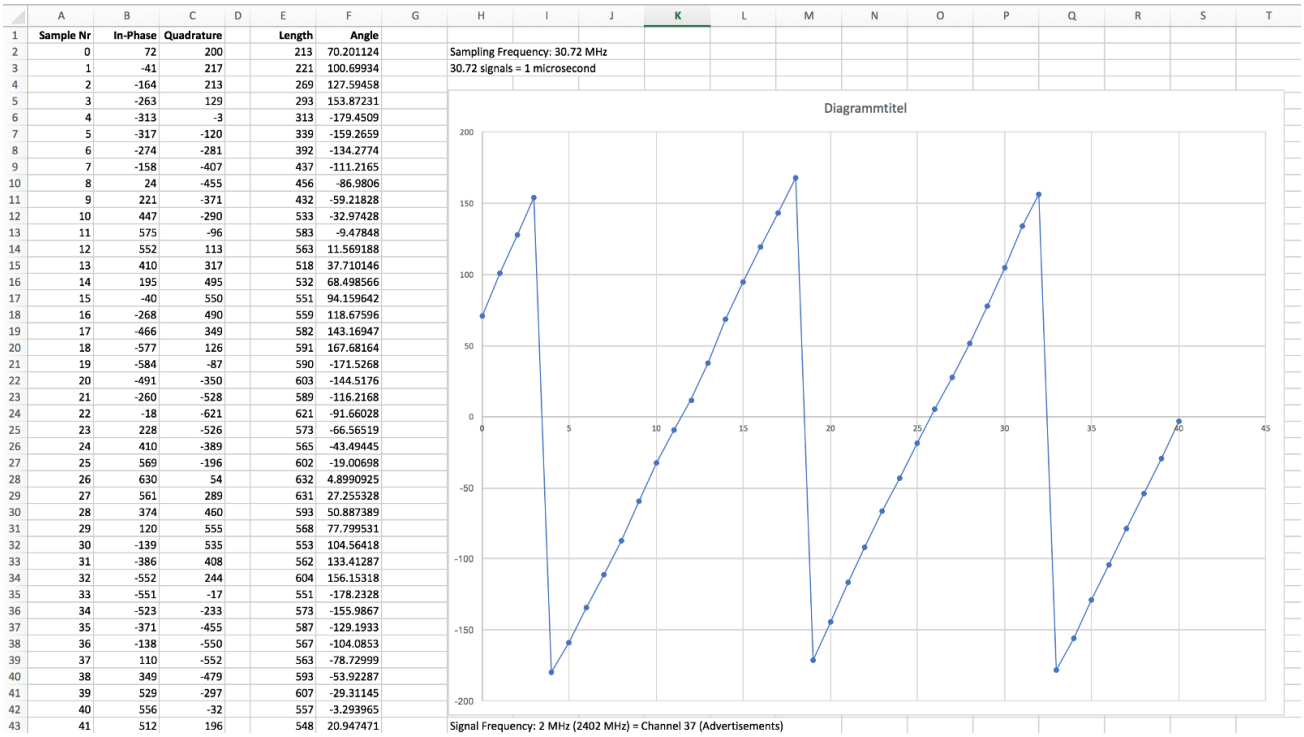
Baseband Processor

As mentioned above, the baseband processor accepts as input the output from the AD9364 analog-to-digital converter (ADC).

The ADC has two output channels, the in-phase and the quadrature channel, each with a resolution of 12 bits (-2048 to 2047). Together, these two values represent a vector that corresponds to the current RF signal that the AD9364 has detected.

Data is delivered at a rate of 30.72 MS/s (million samples per second; the maximum that the AD9364 can deliver is 61.44 MS/s, but I've reduced the data rate for the moment.)

Let's have a look at a small subset of one of the recorded sets of samples:



Sample Nr	In-Phase	Quadrature	Length	Angle
0	72	200	213	70.201124
1	-41	217	221	100.69934
2	-164	213	269	127.59458
3	-263	129	293	153.87231
4	-313	-3	313	-179.4509
5	-317	-120	339	-159.2659
6	-274	-281	392	-134.2774
7	-158	-407	437	-111.2165
8	24	-455	456	-86.9806
9	221	-371	432	-59.21828
10	447	-290	533	-32.97428
11	575	-96	583	-9.47848
12	552	113	563	11.569188
13	410	317	518	37.710146
14	195	495	532	68.498566
15	-40	550	551	94.159642
16	-268	490	559	118.67596
17	-466	349	582	143.16947
18	-577	126	591	167.68164
19	-584	-87	590	-171.5268
20	-491	-350	603	-144.5176
21	-260	-528	589	-116.2168
22	-18	-621	621	-91.66028
23	228	-526	573	-66.56519
24	410	-389	565	-43.49445
25	569	-196	602	-19.00698
26	630	54	632	4.8990925
27	561	289	631	27.255328
28	374	460	593	50.887389
29	120	555	568	77.799531
30	-139	535	553	104.56418
31	-386	408	562	133.41287
32	-552	244	604	156.15318
33	-551	-17	551	-178.2328
34	-523	-233	573	-155.9867
35	-371	-455	587	-129.1933
36	-138	-550	567	-104.0853
37	110	-552	563	-78.72999
38	349	-479	593	-53.92287
39	529	-297	607	-29.31145
40	556	-32	557	-3.293965
41	512	196	548	20.947471

Raw samples from the AD9364 and visualization in Excel

The ADC data is in columns B and C. Column F and the diagram contain the angle of the I/Q vector in the cartesian I/Q coordinate system, calculated by Excel.

Notice that the vector rotates from about -180 degrees at sample 5 to about +180 degrees at sample 19 and then wraps around for the next rotation. The time between two samples is about 32.5 nanoseconds, so 15 samples take about 0.5 microseconds which is the time it takes for the I/Q vector to complete a full rotation. 0.5 microseconds correspond to a frequency of 2 MHz, i.e. the signal that we can see has a frequency of 2402 MHz.

If you remember the channel diagram from part 1, you know that 2402 MHz happens to be the advertising channel 37. So even without any further processing, we seem to see a transmission on this channel with our eyes.

Input and output of the Baseband Processor

The task of the baseband processor is to take the I and Q values and generate a bitstream output for each channel:

What we ignore for the moment is how to extract BLE packets from the bitstream. That should be pretty straight-forward and will be the topic of the next part. It involves detecting the start of a packet, de-whitening the packet data and verifying the packet data with the CRC checksum.

I've already revealed in part 1 our coarse strategy for generating the bitstreams. Let's repeat it and add some details. At the end of this part, I'll also give an overview of the alternatives that we've considered and what made us go with the approach we did.

Channelization of the ADC samples

The data stream from the AD9364 contains multiple channels at once. So, what we do first is to extract individual channels from the data stream. To that purpose, we mix the data stream with a number of clean carriers at frequencies 2 MHz, 4 MHz, 6 MHz and so on and low-pass filter the mixing result. From this mixing and filtering, we get one new stream per channel. What we gain is that every one of these streams now has a center frequency of zero, i.e. the subsequent processing steps can be identical and FPGA resources can be re-used (e.g. via pipelining) for all channels.

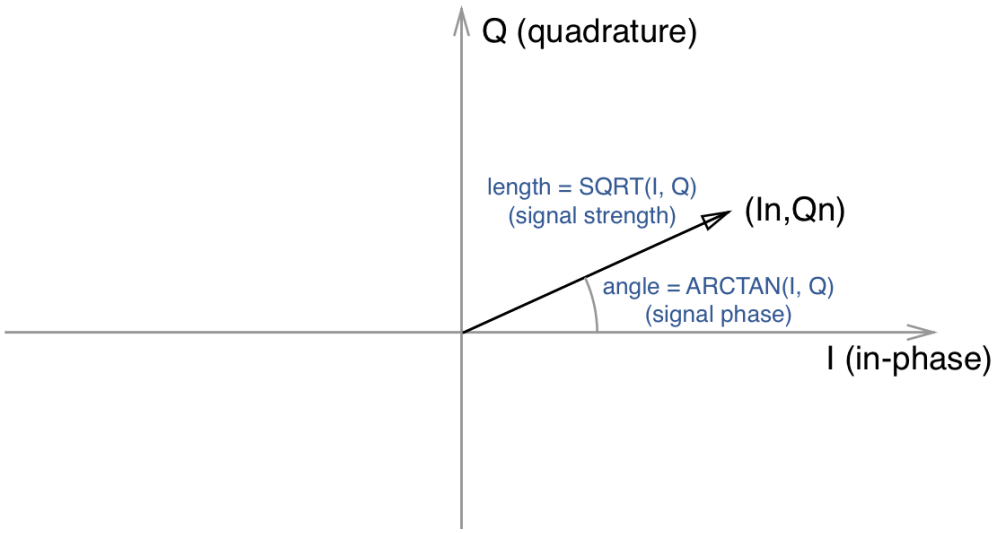
In the excel diagram at the beginning of this post, we can see the angle of the I/Q vector over time. The vector rotates quite fast, which makes it difficult to see minor differences in the rotation

modulated, i.e. shifted up and down by about 180 kHz. The result is that the frequency for the first BLE channel switches between 2401.82 and 2042.18 MHz. Moved down by 2.4 GHz into our “baseband”, the frequency switches between 1.82 and 2.18 MHz.)

I/Q vector rotation of each channel

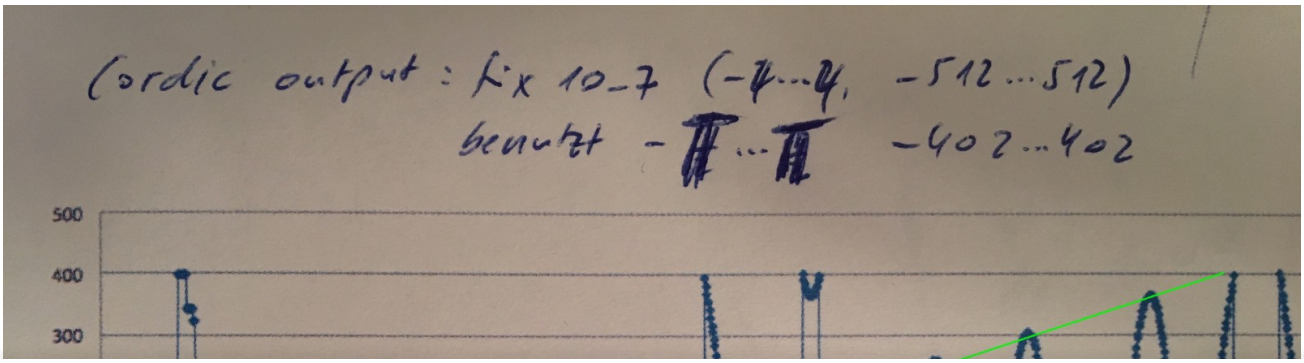
After moving the channel center frequencies to zero (DC) by multiplying the channels with the sine waves of 2 MHz, 4 MHz and so on and lowpass-filtering the result to get rid of unwanted mixing products, we can apply an ARCTAN function using a CORDIC IP core from Xilinx to convert from the cartesian I/Q coordinates to a polar coordinate system.

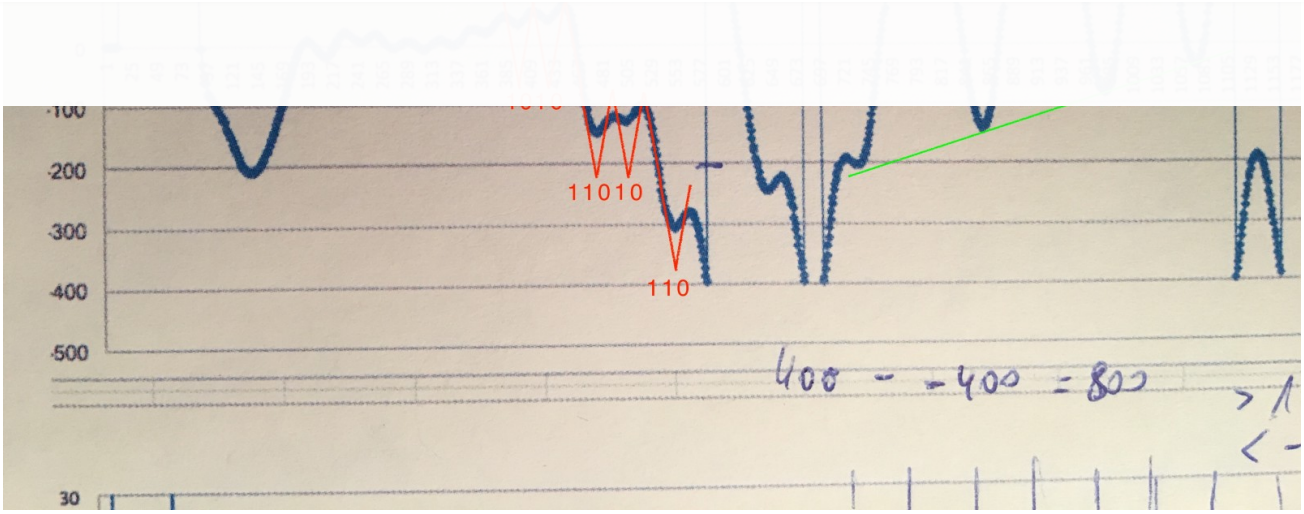
The length of the vector corresponds to the signal strength, but we’re more interested in the angle of the vector:



Cartesian (I,Q) and polar representation of a sample (In, Qn)

After applying the ARCTAN function, the modulation is already quite easy to see. The following image is from the same set of samples:





The thick blue line that we can see in the diagram is the measured *angle* of the signal vector over time. It varies between -402 (-π, -180°) and +402 (+π, +180°) and wraps around at the boundaries (e.g. near sample 577) as the vector performs a full rotation.

The angle increases if the signal frequency is positive and decreases for negative frequencies. In BLE, a frequency of -180 kHz (with respect to the channel center frequency) corresponds to a *zero* bit and a frequency of +180 kHz to a *one* bit. *(Sidenote: the y axis is mirrored in the diagram.)*

I've added a red curve for a small number of bits to show how the angle would look without Gaussian filtering in the modulation and with a perfect receiver and transmitter. In practice, the Gaussian filtering smoothens the triangular shape, and imperfections in the transmitter and receiver frequencies cause a drift (in this case an upward drift as indicated by the green lines). The BLE standard accepts a frequency deviation of up to +/- 150 kHz for the center frequency, so that's expected and we need to be able to handle it in the baseband processor.

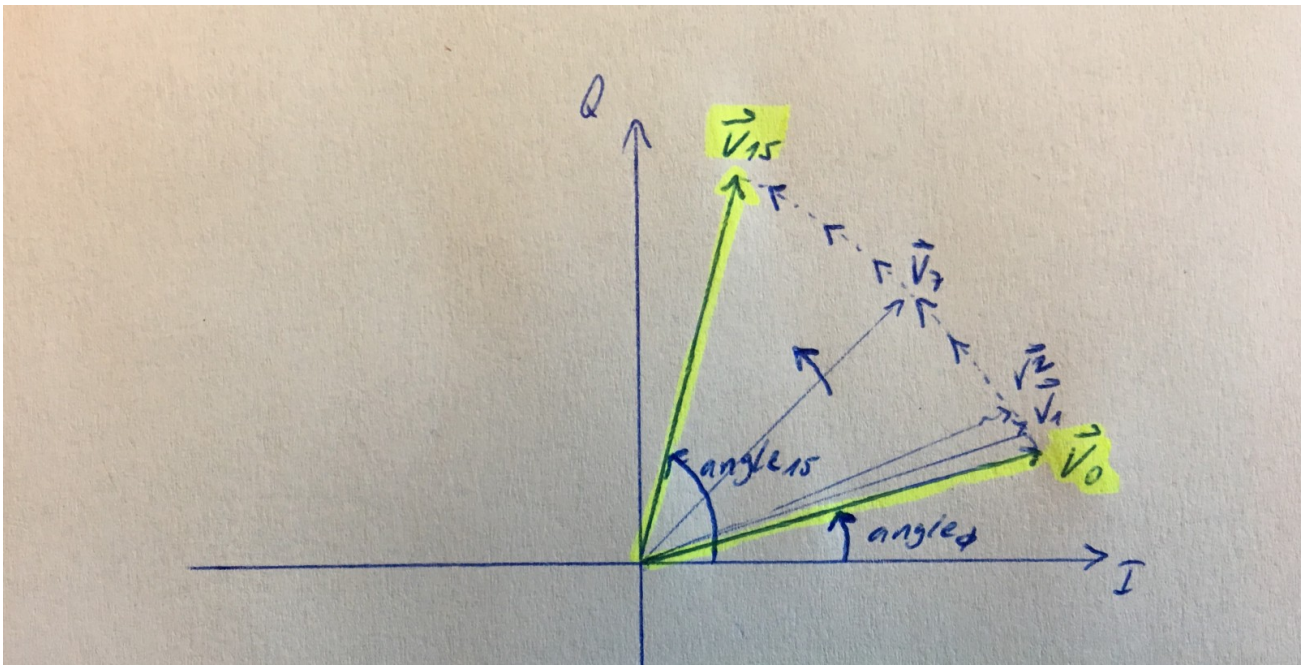
Let's have a more detailed look at the numbers. You may want to skip this section if you're not interested in the details…

First, how far off is the receiver frequency from the transmitter frequency? In the example above, the drift is approximately 100 units (about 45°) in a 16 microseconds window (samples 769 to 1015), i.e. 7'800 rotations per second, which corresponds to a frequency difference of 7.8 kHz between the transmitter and the receiver.

At +180 kHz for a logic one, the vector rotates slowly and takes about 5.6 microseconds for a single rotation (180'000 rotations per second). A full bit in BLE takes only 1 microsecond to

The red line going down corresponds to a logic one being transmitted and measures about 143 units on the y axis (from a total span of about 804 units, corresponding to 64° from the full 360°).

We're sampling at 15.36 MHz, i.e. in intervals of about 65 nanoseconds, so with 64° in 1'000 ns, we notice that the vector rotates by only 4.16° per sample, or more specifically, by +4.16° for a logic one and by -4.16° for a logic zero.



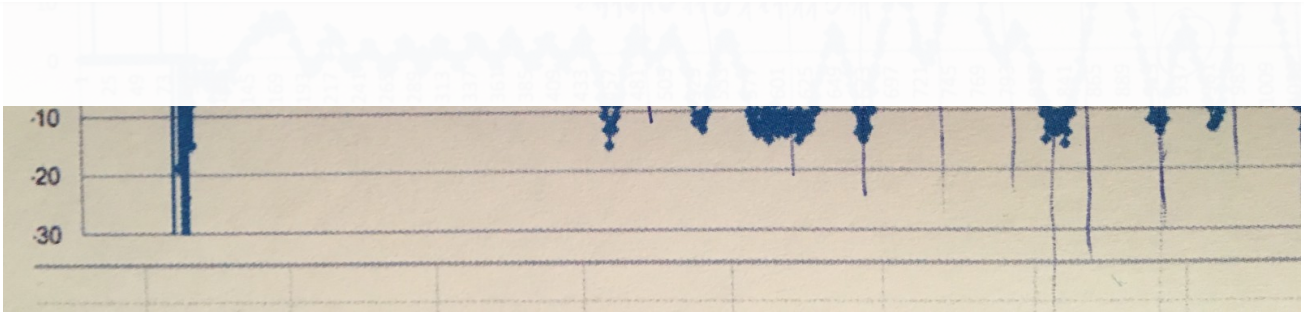
Change of the signal vector angle during a single bit (64°, 15 samples, 1 microsecond)

Between samples 361 and 450, the signal toggles a couple of times between zero and one, followed by two ones, the sequence 010 and another two ones between samples 520 and 550.

The wiggly line between samples 193 and 433 reflects the preamble of a BLE advertising packet. According to the standard, the preamble consists of 8 bits 01010101. As we can see, it can also be longer in practice (about 16 bits in this case).

First derivative of the signal vector angle

To detect the zeroes and ones, we differentiate the signal with respect to the time because we're interested in the *change of the angle over time* (e.g. +4.16° per sample for a logic one, -4.16° per sample for a logic zero). This sounds complicated, but all we do is to take the difference of two samples, i.e. y(n) = x(n)—x(n-1). The resulting graph looks like this:



Notice that the upward trend has disappeared! *(Sidenote: That's not entirely true: by differentiating the signal, the upward trend was converted to a constant offset in y towards positive y values… but anyway, the graph looks much more symmetric now.)*

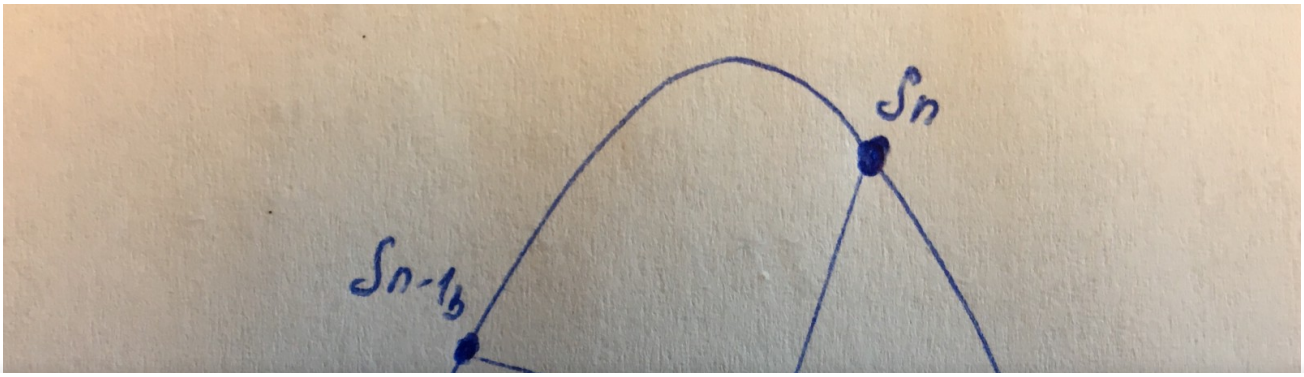
Notice also that it's no longer the slope which defines a zero and one but the absolute value.

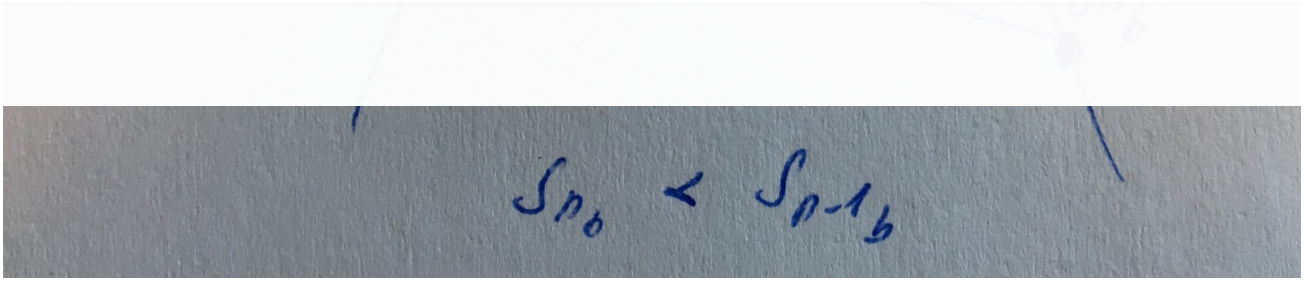
From this graph, a human can easily identify the individual bits. I've annotated the bits of the advertising channel packet address.

Symbol timing recovery

Unfortunately, the FPGA lacks any sense of intuition, so we need to substitute this lack with an algorithm. The algorithm needs to be able to "lock in" to the bit sequence, i.e. it needs to be able to detect the optimum position for sampling. It also needs to be able to adjust itself over time as the timing in the transmitter and the receiver may not be equal.

A short Google search reveals multiple ways to achieve this goal. One way which is often choosen is to use the *Gardner algorithm* for timing recovery (see Gardner, F. M., "A BPSK/QPSK Timing-Error Detector for Sampled Receivers", *IEEE Transactions on Communications*, Vol. COM-34, №5, May 1986, pp. 423–429). This algorithm requires two samples per period to determine if the sampling is early or late. With the early/late information, we can adjust the sampling time:



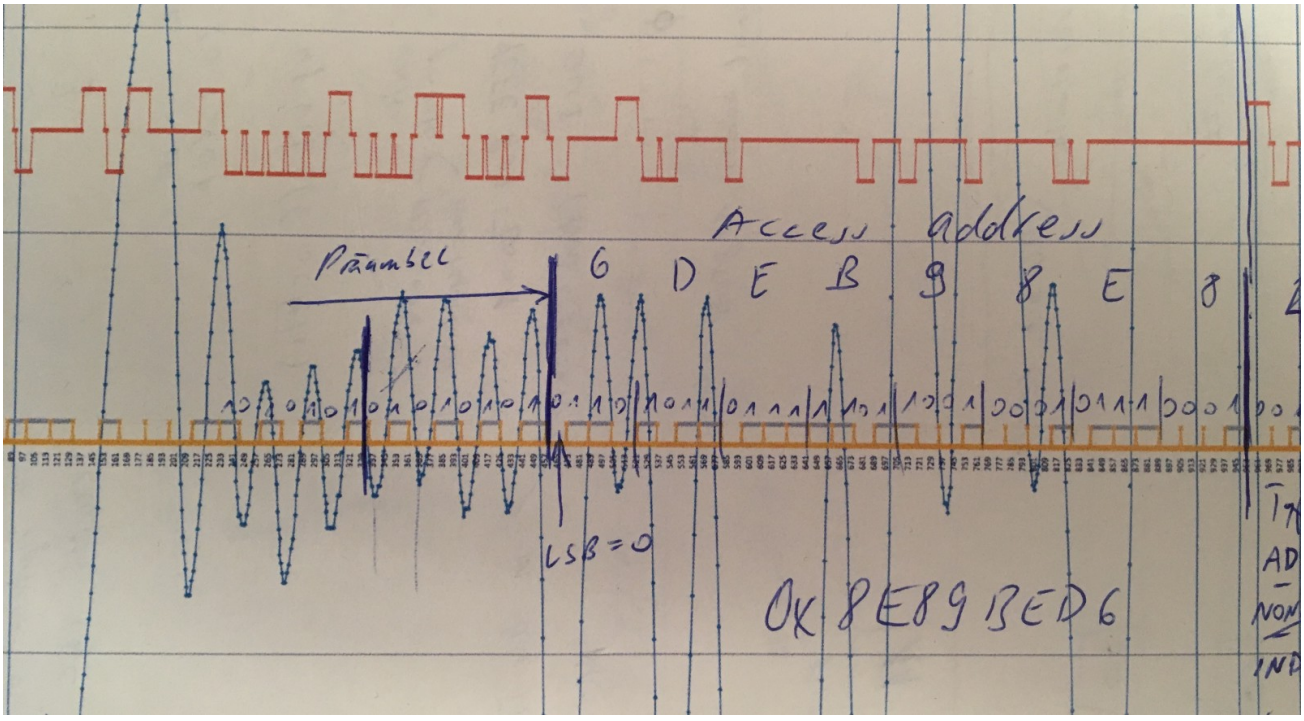


Gardner algorithm: The sampling happens too late in this example, so the slope of the intermediate samples is negative. Control logic should sample the next bit at a slightly earlier time.

The Gardner algorithm calculates the slope of intermediate samples to see if the sampling happens too early or too late. Based on this result, the control logic can adjust the delay before the next sample is taken.

In our case, the delay is fixed as samples arrive at 15.36 MHz. Thanks to the large oversampling that we use (factor 15.36, because the BLE bit rate is 1 MHz), we can simply consider an earlier or later sample.

The red line in the upper area of the next diagram is the number of samples that define one bit for us, corrected by the output from the Gardner algorithm. If the Gardner algorithm says we're late with sampling, we reduce the number of samples from 15 to 14. If we're early, we go to 16:



Applying the Gardner algorithm for symbol timing recovery

With all these processing steps, the baseband processor is able to correctly derive the bit stream for the beginning of this BLE advertisement packet (yellow / gray line on the x axis).

Summary

Let's summarize again what we did to achieve this goal:

- Generate clean sine waves at frequencies 2 MHz, 4 MHz, 6 MHz, …
- Mix the ADC output (30.72 MHz, 2x12-bit I/Q signal) individually with each sine wave to center every channel at 0 MHz (DC).

Then, for every channel:

- Lowpass-filter every channel (filter bandwith about 500 kHz) to remove unwanted mixing products.
- Calculate the arc tangent of the I/Q vector to get the rotation angle of the vector.
- Differentiate the rotation angle to get the rotation speed, which corresponds to the frequency offset from the center frequency (+180 kHz, -180 kHz for BLE signals).
- Lowpass-filter the differentiated angle to remove noise.
- Apply the Gardner algorithm for timing recovery.

With a reasonably good looking bit stream, we're ready for the next part where we'll try to extract BLE packets from the bit stream and send them to the host computer to analyze them e.g. in Wireshark.

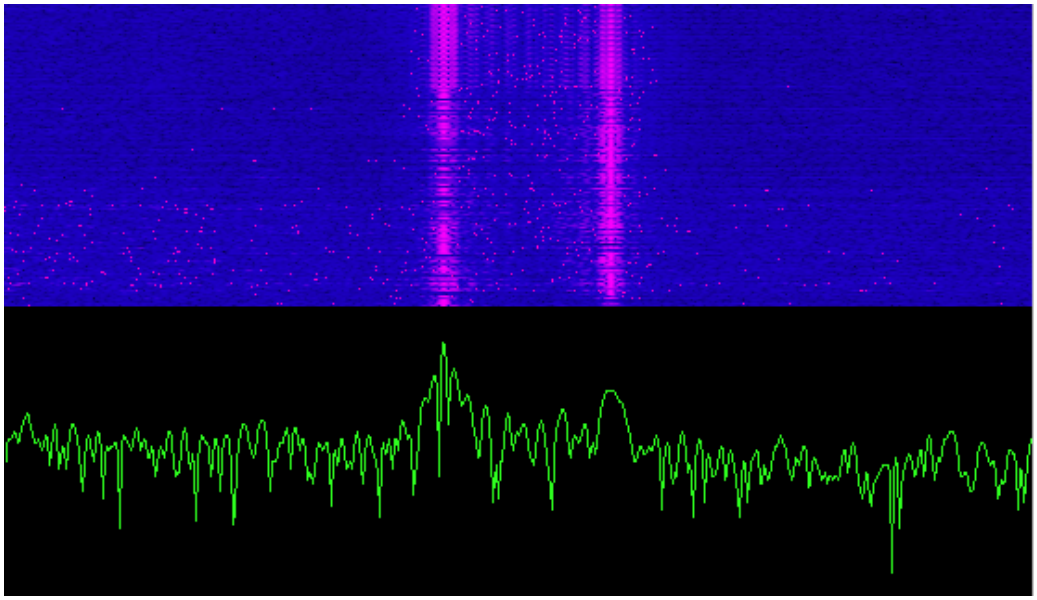
Before we finish for today, let's have a look at some alternatives we've considered for the implementation of the baseband processor and why we've dropped them.

Alternative approaches

Fourier analysis

Our first and most obvious idea was to recover the bit streams by applying a Fourier transformation on the ADC data stream and measure the intensity in the relevant Mark and Space bins for each channel.

You may be familiar with the typical waterfall diagram that some RTTY and FSK software packages generate. The following is a short recording of the German weather service DDH47 in Pinneberg.



Audio spectrum of the DDH47 weather service transmitter DDK 2 on 4583 kHz

I've created this one myself—you find much better ones on YouTube, for example here.

The DDK 2 transmitter on 4.583 MHz uses a 50 Bd RTTY signal. It shifts the frequency by +/- 225 Hz to represent 50 Mark and Space bits per second, i.e. the two pink lines in the waterfall diagram above are 450 Hz apart from each other.

Applying this idea to BLE, the image would represent one channel, e.g. advertising channel 37 at 2402 MHz, and the two pink lines would correspond to Mark and Space frequencies of 2401.82 and 2402.18 MHz, separated by 360 kHz instead of 450 Hz. For the moment, we ignore the fact that BLE actually uses a **G**FSK (Gaussian FSK) modulation, so, of course the signal wouldn't jump so nicely between the two frequencies but change more slowly.

The reason why this approach doesn't work is that we can't get the necessary resolution in time and frequency at the same time.

Let's do the math:

For a reasonable resolution in the time domain, we'd like to have two evaluations of the Fourer transform per BLE bit, i.e. one evaluation every 500 nanoseconds. Let's say we've configured the

gives 15 samples from the AD9364 for each DFT window. At 15 samples, however, the resolution in the frequency domain is only 0.5*Fs/N = 0.5 * 30.72 MHz / 15 = 1 MHz.

We could increase N to get smaller frequency bins, e.g. to 150 for 100 kHz bins, but as a result, multiple bits (specifically, 5 bits at N=150) are merged into the bin, so it's no longer possible to retrieve the bit stream.

Consequently, we can either have enough bins to detect the Mark and Space signals for each channel or enough time resolution to recover the 1µs BLE bits but not both at the same time.

To fix this issue, we'd need to bandpass-filter the signal and apply the Fourier transform multiple times. This would probably work but consume much more resources. In addition, we'd have to find a way to detect for each channel which bins to use for zeroes and ones, as the transmitter and receiver frequency may deviate by up to 150 kHz from the exact value.

Direct bandpass filtering

If the Fourier transformation doesn't work, let's think about another approach. We could place two narrow bandpass filters at the Mark and Space position of every BLE channel and just sample the strength of the filtered signal, correct?

I believe this approach would actually work, but there are a couple of issues:

- First, it takes a huge amount of FPGA resources to instantiate 80 narrow bandpass filters (2 for each one of the 40 BLE channels). I didn't try it, but I'm not sure the design would fit in the Zynq XC7Z020. Actually, I'm quite sure it wouldn't. (To mitigate this effect, we could move all channels to DC as we do in the chosen approach.)
- Second, we'd have to handle the signal strength and changes of signal strength over time. In the approach that I describe above, we only look at the angle of the (I,Q) vectors and ignore their length (which represents the signal strength). With the bandpass filters for Mark and Space, we'd get only a little bit of energy in the filters for weak signals and very much energy for strong signals and would have to compensate for this effect. (We'd actually have the same issue with the Fourier transform, too.)
- Third, the center freqency may deviate by +/- 150 kHz from the exact value for both the transmitter and the receiver, so the energy likely wouldn't be evenly distributed between the filters. We'd also need to compensate for this effect.

Thanks to Alex Suzuki.

Ble Fpga Soc

One clap, two clap, three clap, forty?
By clapping more or less, you can signal to us which stories really stand out.

20 1

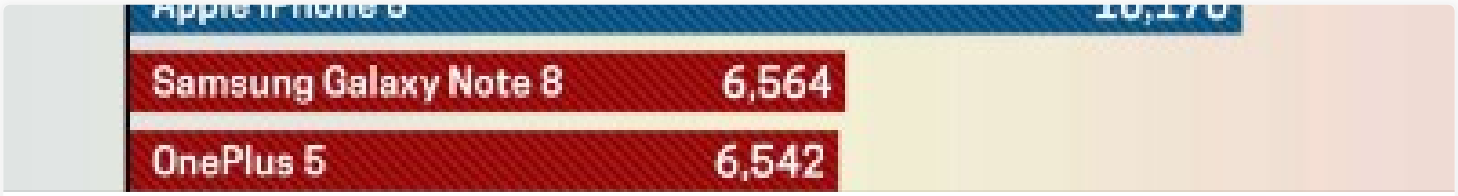
Andreas Schweizer Follow

Classy Code Blog Follow
Tech babble and random rants

More from Andreas Schweizer

Implementing a CPU in VHDL — Part 2

Andreas Schweizer
12 min read



Also tagged Soc

Apple’s Chipnomics — How is apple chip faster than the rest

4 min read

Also tagged Ble

Airbitz Enables BLE-driven Wireless Payments for iPhone and Android

Edge
2 min read

1

Responses

Show all responses