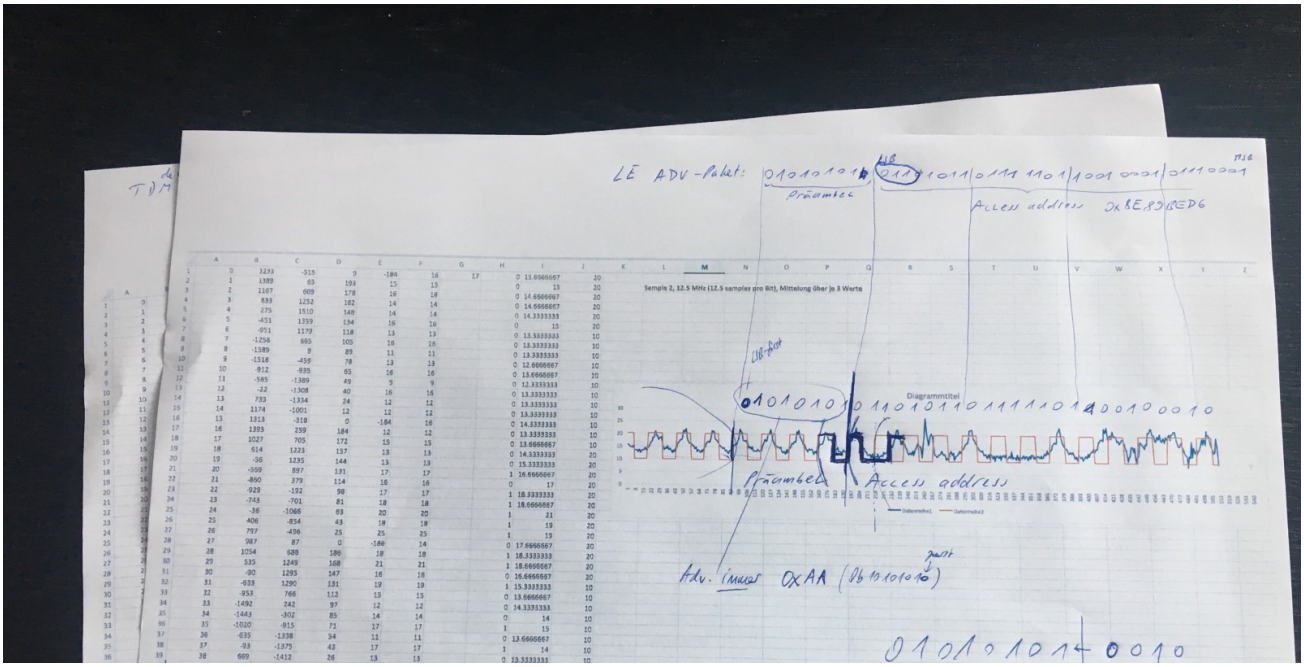
Andreas Schweizer
Jun 5, 2017 · 8 min read

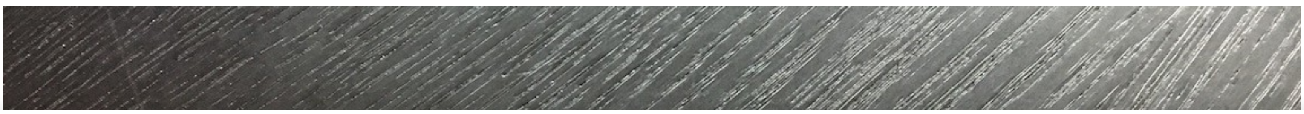
# On Developing a Wideband BLE Sniffer—Part 1

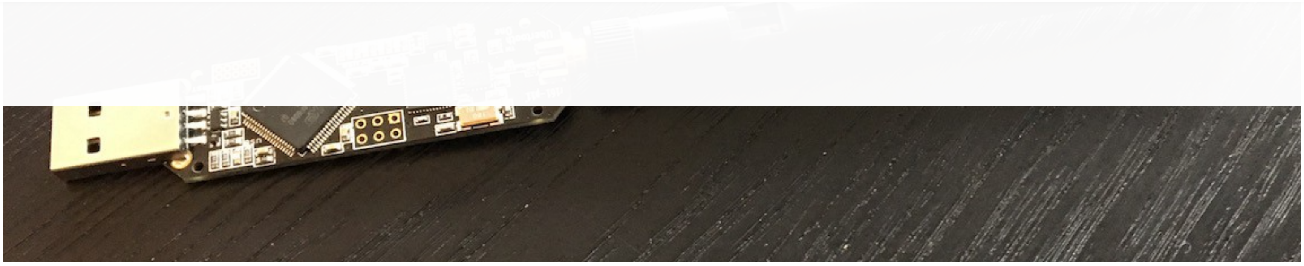



From time to time, we need to have a detailed look at BLE traffic. One would expect that it's easy to find a device on the market that can monitor the whole BLE traffic at once and just dump all packets that it receives, but unfortunately, that doesn't seem to be the case…

If we search for BLE sniffers on the web today, we can find two main categories:

- One category contains simple single-channel sniffers that try to capture connections by following the channel-hopping (see below). The devices in this category are relatively cheap but limited in that they often fail to follow the hopping sequence and/or miss some packets. A well-known member of this category is the Ubertooth one:



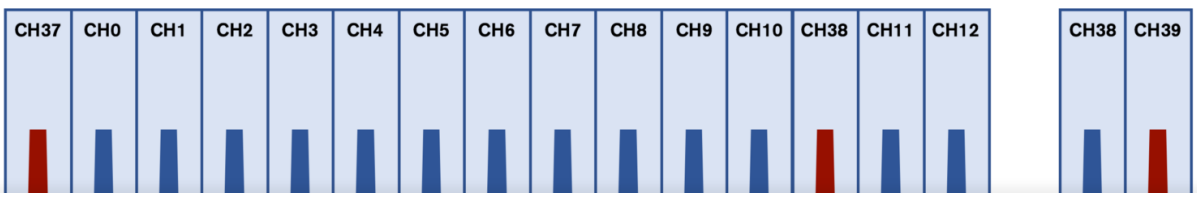


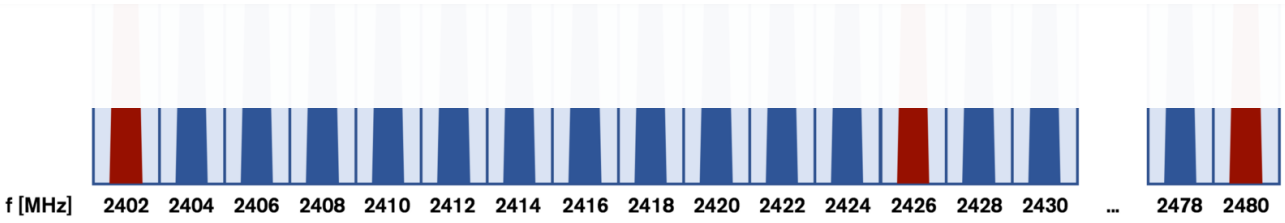
The Ubertooth one

- The other category are professional, high-performance sniffers that monitor all channels at once, support BLE but also classic Bluetooth, support reception but also transmission. These are great, but come at the cost of a medium-size car!

Why is that the case?… There are a couple of aspects that make it more difficult to capture BLE traffic than one might expect:

- BLE spans 80 MHz in the frequency spectrum, split into 40 individual channels. There's no one single frequency a receiver can tune in to and receive the BLE traffic as, for example, in FM broadcasting. Three channels (37, 38, 39) are used for *BLE advertising* where devices broadcast short packets to let other devices know they exist and to transmit small amounts of data. The other 37 channels (0 to 36) are used for bi-directional communication after two devices have established a *connection* between them.
- Once two BLE devices are connected to each other, they apply a *frequency hopping* algorithm based on parameters that they exchange at the time when they initially establish the connection. In other words, they jump from one channel to another in short time intervals. A single-channel receiver needs to chase them to be able to follow the data exchange.
- The BLE data rate is 1 Mbit per second, i.e. a single bit takes only 1 microsecond. (BLE 5 allows for an even higher data rate of 2 Mbit per second). Worst case, with concurrent traffic on all 40 channels at once, the resulting data rate would be 40 Mbit per second. What makes things even worse, to detect a single bit, we likely need to oversample the signal, so even without applying any further processing on the bit streams, we already need significant computing resources just to detect the bits.






BLE Channel Frequencies

This is the first in a series of stories on our quest to figure out if it's possible to create "something in between"—a device that can capture all BLE channels at once, but doesn't need to support classic Bluetooth and doesn't need to transmit anything.

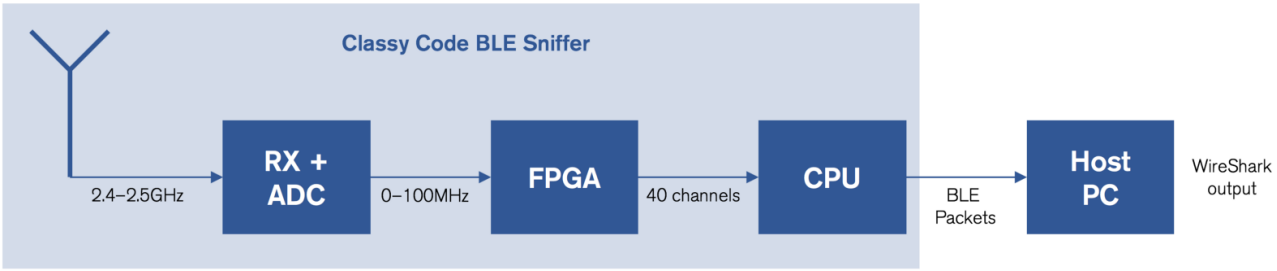
# The Plan

To capture all 40 BLE channels at once, a naïve approach would be to combine 40 individual receivers such as the Ubertooth one and assign a fixed channel to each one of them. I guess this would work but the device would look awful (just imagine the 40 USB connectors… ;-)

The more elegant approach—which we'll take, of course—is to use a single, wideband receiver that spans all channels and try to extract the individual channels from the wideband input signal.

## System overview

From a high-level perspective, this is the system we want to build:




The antenna on the left side of the diagram receives all BLE traffic in the 2.4 GHz ISM band and delivers it to the receiver front-end (first blue box).

The *receiver front-end* (RX + ADC in the image above) performs the down conversion and initial filtering of the BLE traffic into baseband (0 to 80 MHz) for further processing. A fast ADC (analog-to-digital converter) samples the baseband signal which is then digitally filtered and provided to an FPGA (field-programmable gate array, second blue box).

The *FPGA* is responsible for extracting the 40 individual BLE channels, detecting BLE traffic in the channels, recovering the BLE packets on all channels and providing them to a CPU for further processing. *(Could we skip the FPGA and directly feed the data stream to a CPU? Theoretically yes, but in practice, we likely can't find a CPU powerful enough to perform live signal processing on a 80 MHz data stream (3 bytes per sample results in 240 MBytes/s or a total of 2 Gbit/s); consider that the processing includes numerous digital filtering operations and demodulation algorithms including multiplications, and that proper timing is an important factor in recovering the channel bit streams. We could of course record the stream and decode it offline later, but hey, where's the fun then? ;-)*

The *CPU* collects the BLE packets from all 40 channels, optionally post-processes them and combines them into a single data stream that it delivers to a host PC to display it in Wireshark or that it can store on eg. an SD card.

Combining an FPGA with a CPU gives us a lot of flexibility in deciding which part of the processing to perform in hardware and which part in software.

### Functional prototype

The system described above isn't something we can pull from a hat overnight. Instead, it's more like a journey in which we need to learn for each part of the system the options to implement it, decide which option to choose, see if it works and improve it if necessary.

We plan to publish a number of follow-up posts to document this journey and would be glad if you joined us by reading and commenting along the way!

To quickly get first results, we'll start with a reduced functional prototype of the system. It's not the aim of the prototype to be perfect; we're totally happy if we find a working solution which we can optimize later. For example, it's enough if the prototype captures a subset of the 40 channels. 8 or 10 is ok if we see that and how the approach can be extended to the full 40 channels.

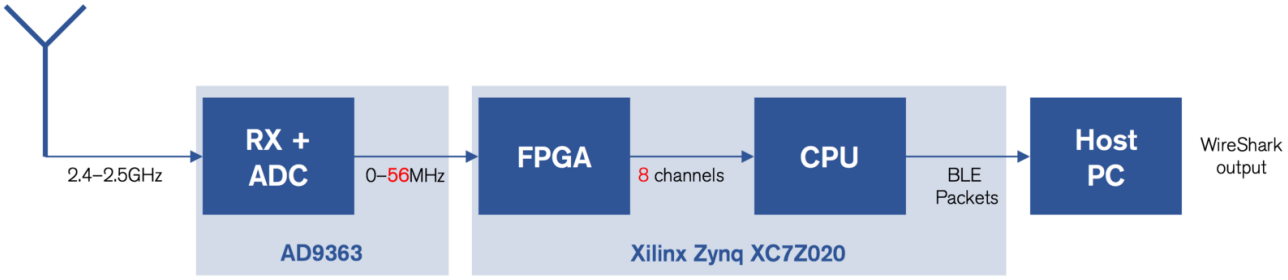
To save time, we'll use existing development boards for the prototype.

## Developing the Prototype

### Platforms

We've decided to use the AD9363 integrated circuit from Analog Devices on an AD-FMCOMMS3-EBZ reference board as wideband receiver front-end for the prototype. The

not wideband enough to cover all 80 BLE channels (we would need two of them as the chip is specified for 56 MHz bandwidth), but it's available at a reasonable price, easily connects to the FPGA board of our choice and can later be used for other RF experiments.

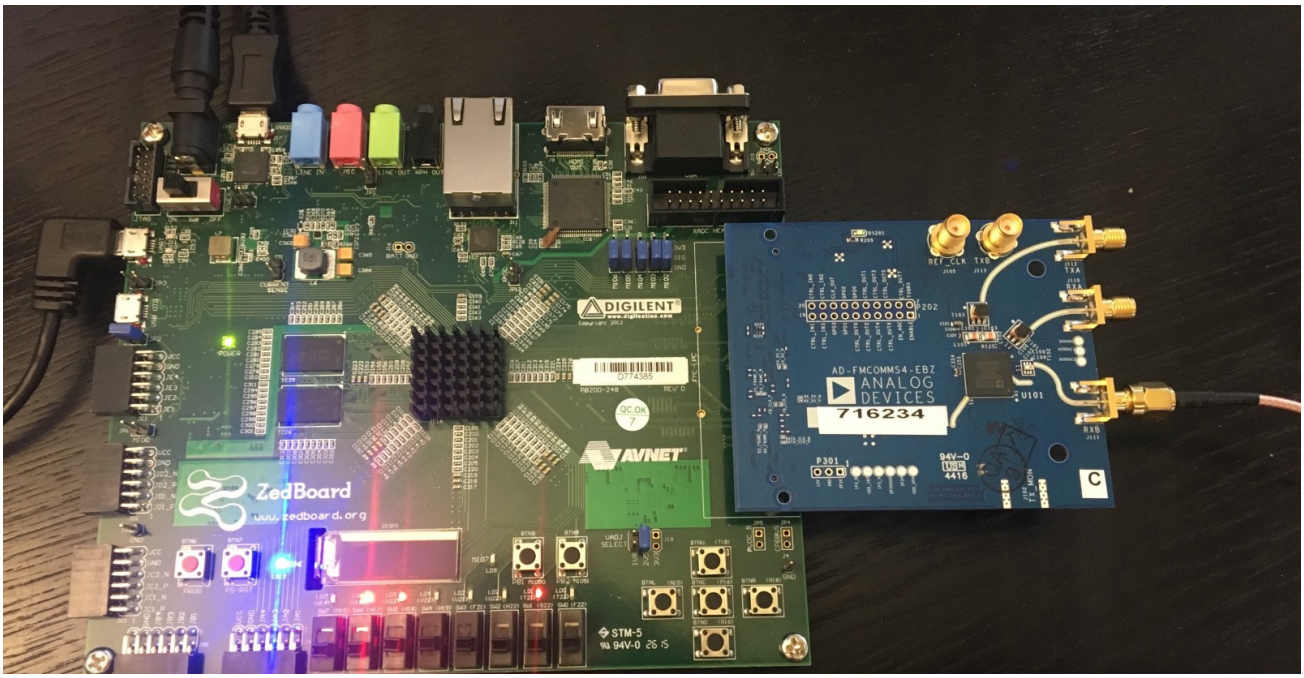



Prototype System Overview

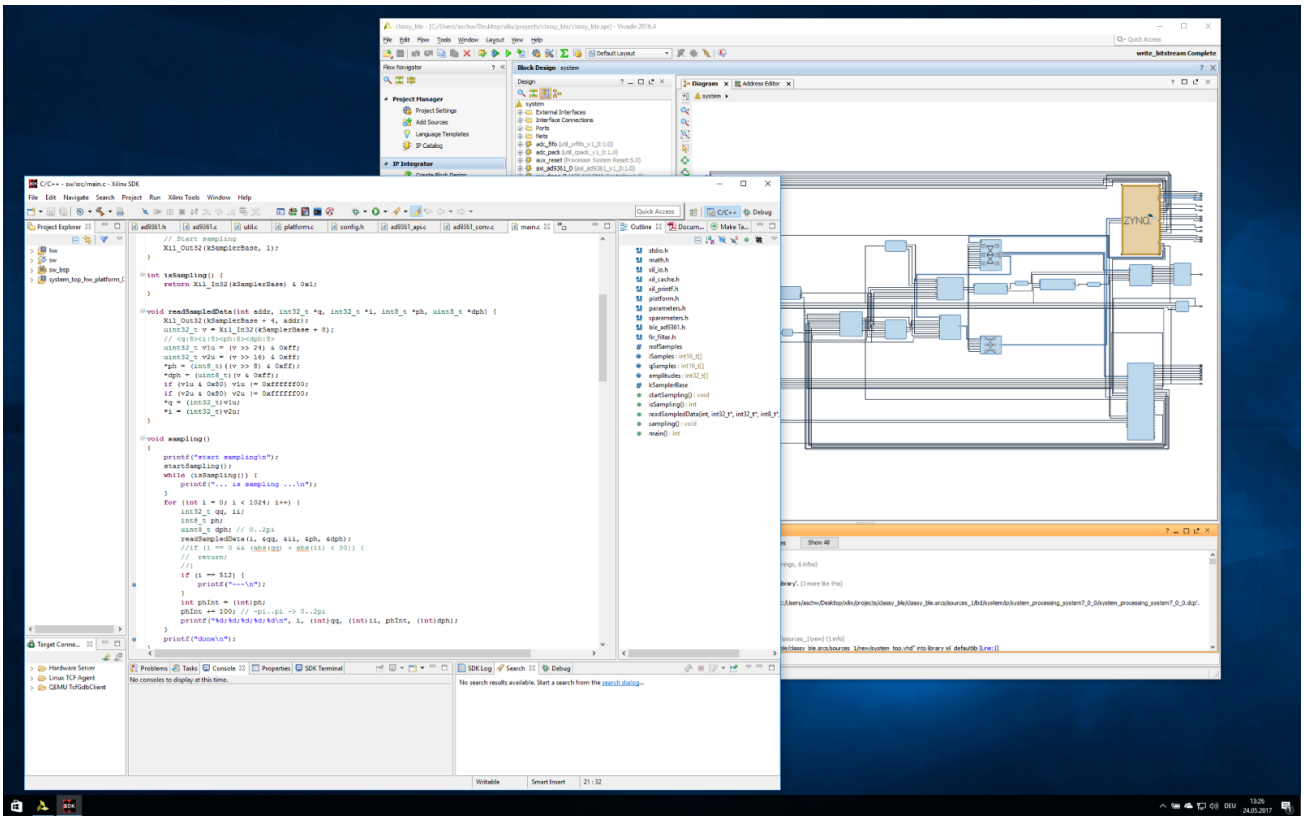
The output of the AD9363 is a digital interface where it delivers the signal samples at a maximum rate of 61.44 MSPS (million samples per second).

For the FPGA, we go with a Xilinx Zynq device. We use the ZedBoard development kit which comes with a medium-sized Zynq, the XC7Z020. This chip integrates programmable logic (85000 cells, 4.9Mb Block RAM and 220 DSP slices to support digital signal processing—ideal for us!) with a dual-core ARM Cortex-A9 core.

The AD9363 board connects to the ZedBoard via a standard FMC connector. You can see the green ZedBoard on the left side and the blue AD9363 board on the right side in the picture:



The ZedBoard comes with a license for Xilinx' integrated development environment "Vivado" and the software development kit (SDK) for the ARM cores in the Zynq. For the AD9363, Analog Devices provide well-documented IP cores and software libraries that we can use in Vivado and in the SDK. We decided to start with their "no-os" driver which runs on a bare-metal ARM core. (Alternatively, they offer a driver to control the AD9363 from Linux but which we ignore, at least for the moment.)



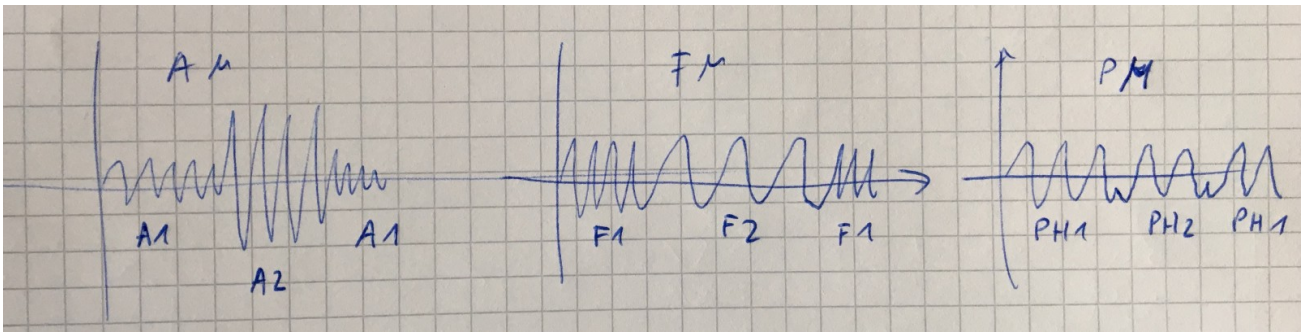
Xilinx Vivado and SDK

It took a couple of days to understand the examples from Analog Devices and set up our own design, but then, we were able to receive raw samples from the AD9363 and print them on the console with one of the ARM cores.

## Channel separation and decoding

The AD9363 delivers signal samples at a rate of 61.44 MHz to the Zynq. Each sample consists of a 12-bit in-phase (I) and a 12-bit quadrature (Q) component. Representing a signal with I and Q components allows to express the amplitude and phase of the signal at the same time.

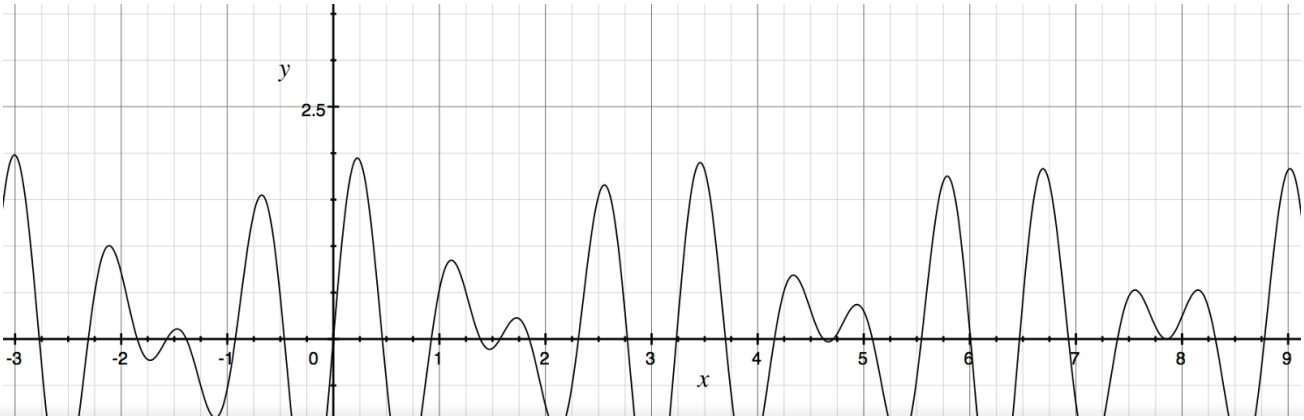
- Amplitude Modulation (AM) changes the amplitude of the signal,
- Frequency Modulation (FM) changes the frequency, and
- Phase Modulation (PM) changes the phase shift of the signal:




Amplitude, frequency and phase modulation of a carrier signal

Bluetooth Low Energy (BLE) uses GFSK modulation, which is short for *Gaussian Frequency Shift Keying* and which is a variant of frequency modulation. The carrier switches between two fixed frequencies to represent a digital “zero” or digital “one”. For example, to transmit a “zero” on the 2402 MHz channel (advertisement channel 37), the transmitter sends a sine wave at about 2401.8 MHz, and to transmit a “one”, it changes the frequency of the sine wave to about 2402.2 MHz. These two frequencies are typically called *mark* (1) and *space* (0). Instead of switching instantaneously between the two frequency (FSK), BLE applies a Gaussian filter to the switching to reduce interference with neighboring channels (GFSK).

The wideband output from the AD9363 contains the information of multiple BLE channels at the same time. For example, if zeroes are being transmitted on the 2406 MHz and 2408 MHz channels at the same time, the signal contains the sum of two sine waves (2405.8 and 2407.8 MHz):



In a first step, the BLE sniffer needs to be able to recover the individual channels from this combined signal. There's more than one way to do that:

1. We could fourier-transform the signal to find out which frequencies are active at a given point in time;
2. We could use narrow bandpass filters, two per channel, to detect the mark and space of each channel;
3. We could mix the signal with fixed frequency carriers, one per channel, to translate them to zero MHz.

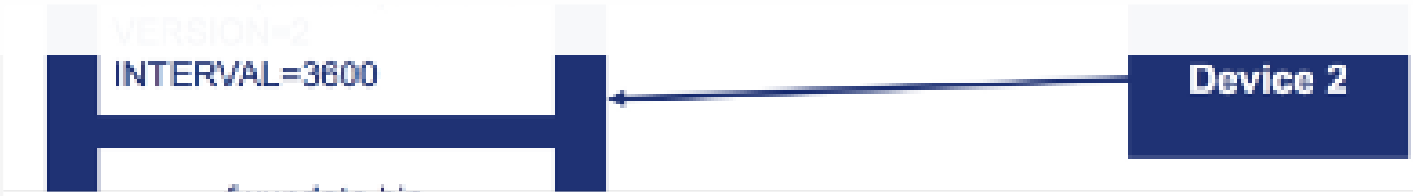
For the prototype, we'll use approach 3. Approach 1 has some issues with the resolution in the time domain, and approach 2 is resource intensive because it processes signals at high frequencies. It also has issues if the transmitter frequency is slightly inaccurate.

In the next blog post, I'll explain these approaches and our solution in more detail.

Are you curious how the journey continues? Leave us a comment and recommend it!

Thanks to Alex Suzuki.

Bluetooth Low Energy | Engineering | Fpga | Rf

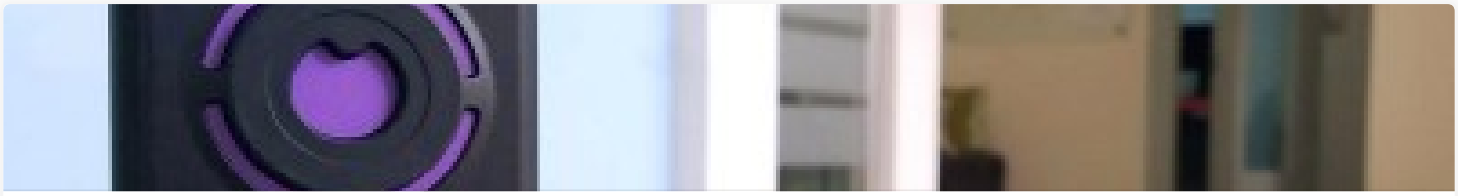



More from Classy Code Blog

## Secure over-the-air updates for ESP32

Andreas Schweizer
6 min read

138

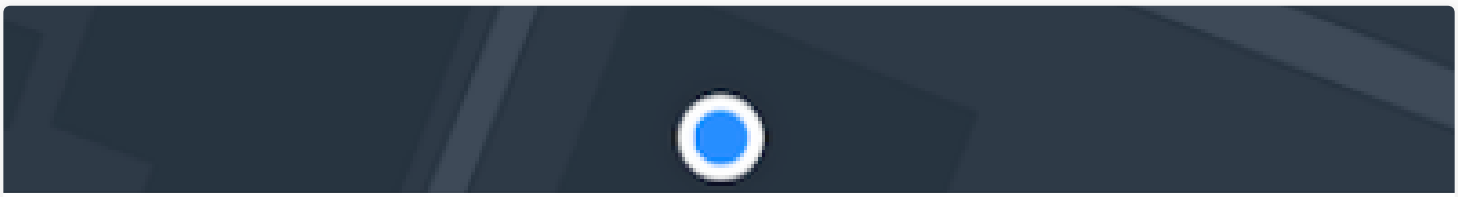


Also tagged Rf

## The Magical and Wonderful World of Walabot!

Adam Benzion
2 min read

3



More from Classy Code Blog

## How to write a Pokémon Go! clone for iOS

Alex Suzuki
7 min read

65

**Responses**

Show all responses