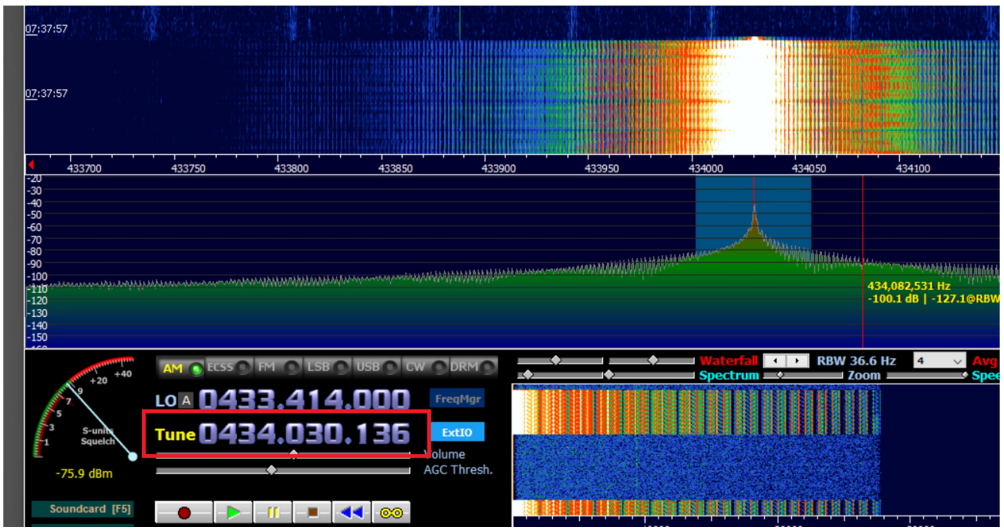
Controlling ASK devices via SDR and Raspberry Pi

NB - As F5OEO states in rpitx's readme: *Before you transmit, know your laws.* ***rpitx*** *has not been tested for compliance with regulations governing transmission of radio signals. You are responsible for using your* ***rpitx*** *legally.* Also see the warnings on this page: https://github.com/ha7ilm/rpitx-app-note

OK, so here's a very informal write up on how to use a SDR and the Raspberry Pi to transmit ASK and/or OOK signals which can be used to control devices (I've used it to trigger doorbells and control RC cars). Of course you might also find other uses for it, I'll attempt to explain the process so that you can take it further.

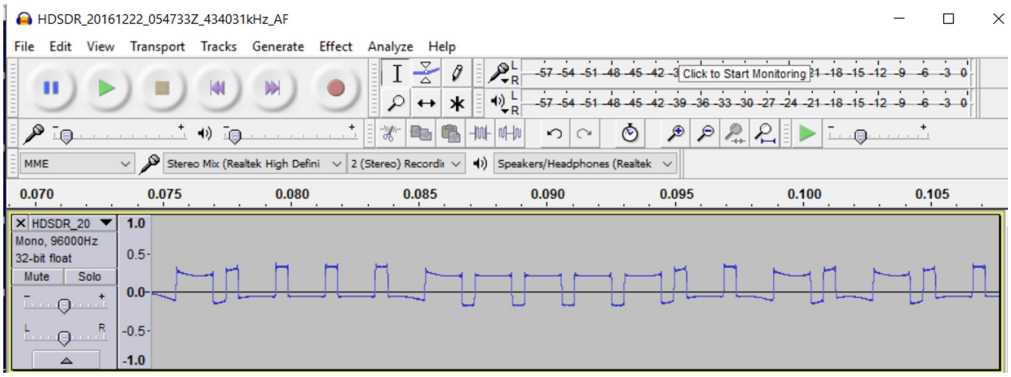
I'm not going to explain things from the ground-up, I'll assume some basic-intermediate experience with SDR and rpitx on the user's part, as well as some programming experience in order to generate the files which you'll feed to rpitx to transmit. Let's get going with the doorbell example.

Step 1: We all know that most doorbells, gate remotes, car remotes etc operate at around 433MHz. All that we need to do is track down the frequency of the transmitter.



So our doorbell is transmitting at 434M030.

Step 2: Let's get a look at the waveform of the signal -> Demodulate and record the signal audio (use AM). I'll be using Audacity to view and analyse the signal. So here's what my doorbell looks like:



Step 3: Find the pulse characteristics. You can do this in one of two ways: The manual approach with Audacity (which you'll anyway need to know for RC cars), or you can use rtl_433 on linux. So we'll quickly do the analysis with rtl_433 (which will actually help because you'll see that our figures between what we derive from Audacity and what rtl_433 gives us match up, roughly at least).

```
root@kali:~# rtl_433 -A -f 434034000

Detected OOK package
Analyzing pulses...
Total count:   24,  width: 11290                (45.2 ms)
Pulse width distribution:
 [ 0] count:   11,  width:   366 [365;368]   (1464 us)
 [ 1] count:   13,  width:   125 [123;129]   ( 500 us)
Gap width distribution:
 [ 0] count:   11,  width:   118 [116;121]   ( 472 us)
 [ 1] count:   12,  width:   359 [357;362]   (1436 us)
Pulse period distribution:
 [ 0] count:   23,  width:   485 [482;487]   (1940 us)
Level estimates [high, low]:  15941,   1080
Frequency offsets [F1, F2]:    -707,     0   (-2.7 kHz, +0.0 kHz)
Guessing modulation: Pulse Width Modulation with fixed period
Attempting demodulation... short_limit: 245, long_limit: 363, reset_limit: 363, demod_arg: 0
pulse_demod_pwm(): Analyzer Device
bitbuffer:: Number of rows: 1
[00] {24} 78 35 ad : 01111000 00110101 10101101
```

So we see that the pulse period = 1940us. Pulse period for a symbol = pulse width+symbol gap. You'll notice that for both 0 and 1 they add up to around the same value (the pulse period), which is to be expected. Pulse period is the amount of time allocated to transmitting a symbol, in Audacity you can measure this by measuring the time from the start of one symbol to the start of the next.

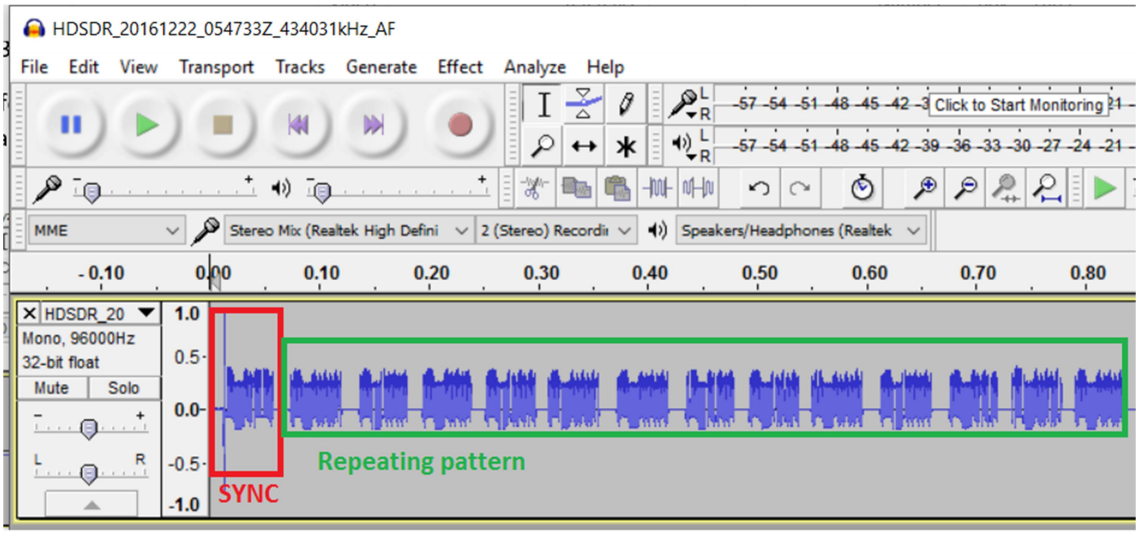
I usually use this command just to see what the transmission pattern is:

```
root@kali:~# rtl_433 -a -f 434034000

*** signal_start = 340375, signal_end = 904311
signal_len = 563936,  pulses = 895
Iteration 1. t: 241    min: 114 (513)    max: 368 (382)    delta 12665
Iteration 2. t: 241    min: 114 (513)    max: 368 (382)    delta 0
Pulse coding: Short pulse length 114 - Long pulse length 368
Short distance: 233, long distance: 1862, packet distance: 4380

p_limit: 241
bitbuffer:: Number of rows: 25
[00] {23} 0f 94 04 : 00001111 10010100 0000010
[01] {24} 87 ca 52 : 10000111 11001010 01010010
[02] {24} 87 ca 52 : 10000111 11001010 01010010
[03] {24} 87 ca 52 : 10000111 11001010 01010010
[04] {24} 87 ca 52 : 10000111 11001010 01010010
[05] {24} 87 ca 52 : 10000111 11001010 01010010
....
```

So from that we can see that the first burst is a shorter sync pattern/preamble, and all the later bursts are just repetitions of the 2nd burst.



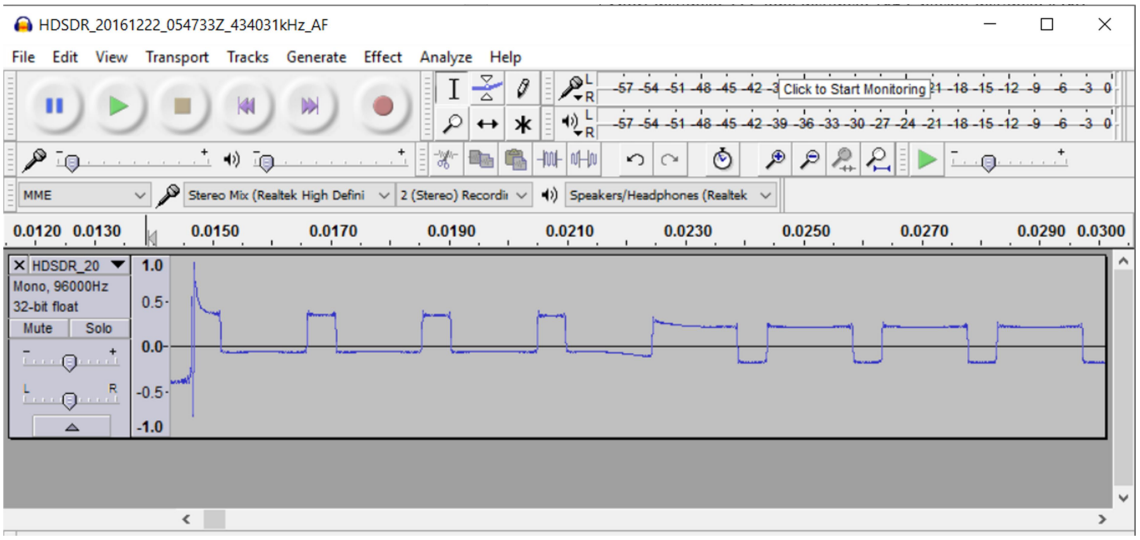
You might have noticed that between the two commands, the bits have flipped for some reason

root@kali:~# rtl_433 -A -f 434034000 -> {24} 78 35 ad : 01111000 00110101 10101101
root@kali:~# rtl_433 -a -f 434034000 -> {24} 87 ca 52 : 10000111 11001010 01010010

Don't worry about this, the important things which you got from these commands are:

1. Symbol period is +/- 1940us
2. Short pulse duration is 500us, its gap duration is 1436us
3. Long pulse duration is 1464us, its gap duration is 472us
4. There are only two unique bursts: The 1st (sync) and the second burst which is just repeated.

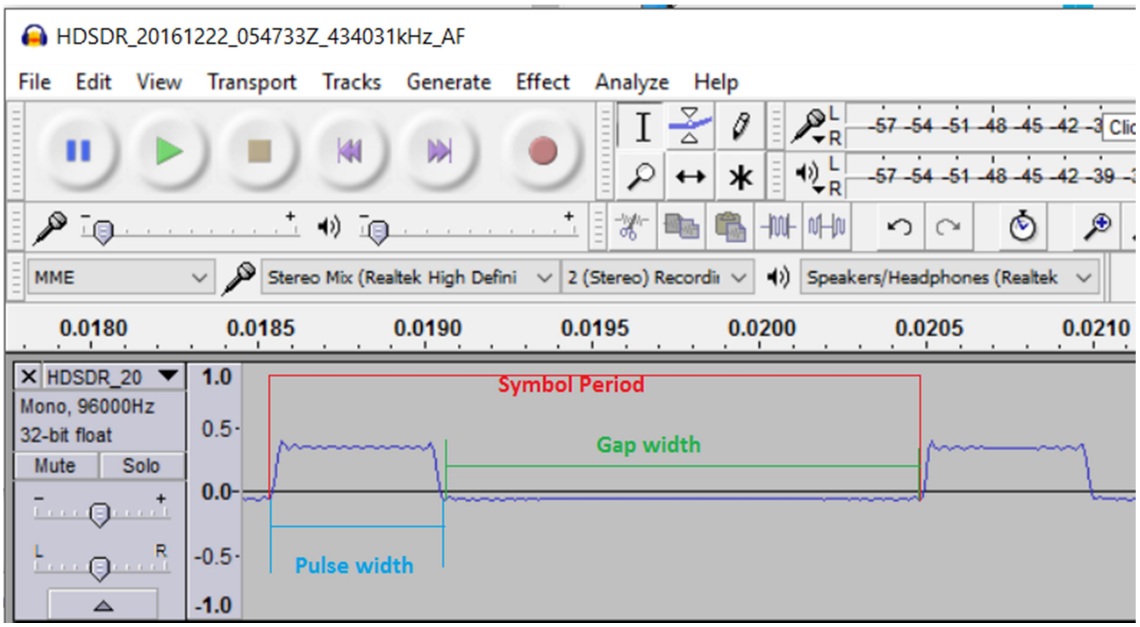
Now here's a close up of the 1st 8 bits of the sync burst



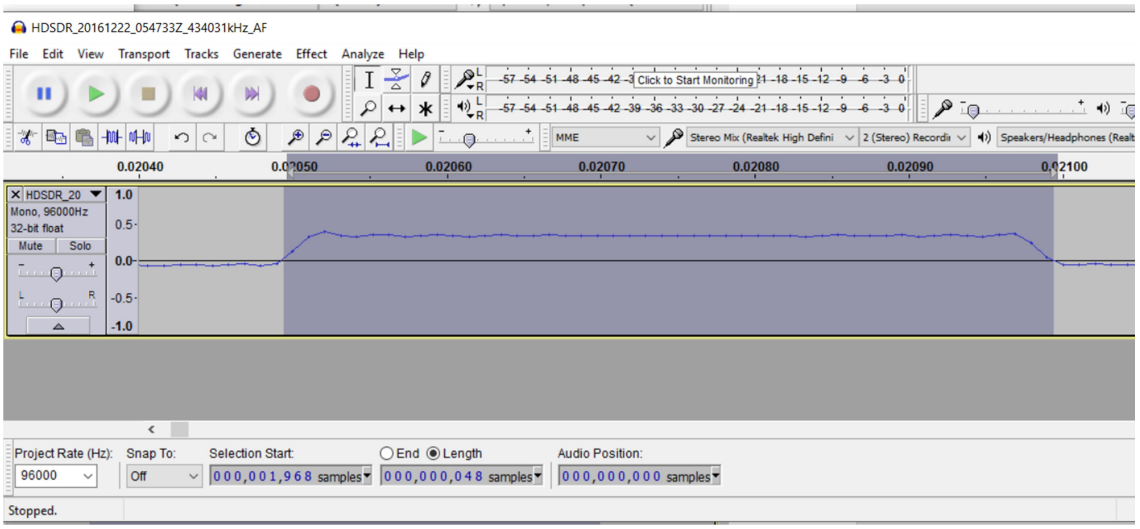
Regardless of what you consider a 1 or a 0, we can see there are 4 symbols of one, and then 4 of the other, which matches what rtl_433 gave us for the first 8 symbols of the sync

[00] {23} 0f 94 04 : 00001111 10010100 0000010

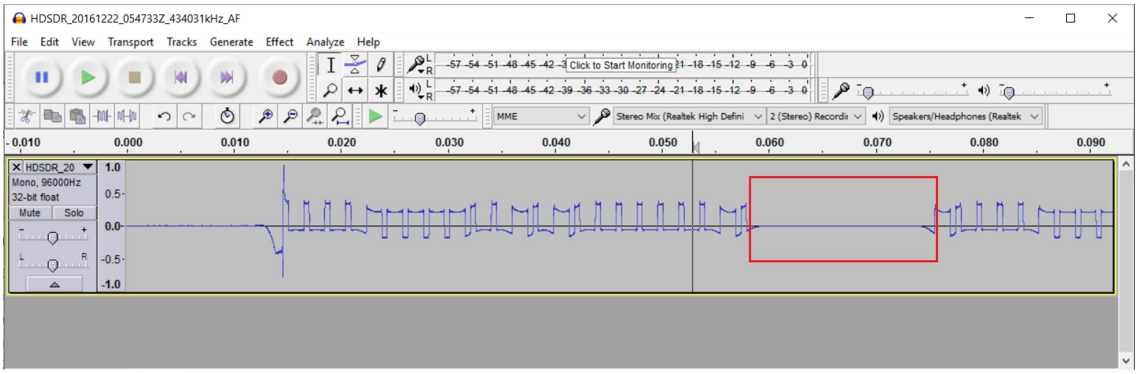
Here's a visual representation of symbol period, pulse width and gap period



How to get the pulse characteristics/timing via Audacity: You can try measure the time directly, but I've found this doesn't work too well for me. I rather use the sample count, and calculate the times.



So the short pulse's pulse width is 48 samples. To get duration (in microseconds), we use (samples/sample rate) x1000000. That gives us (48/96000)x1000000 = 500us, the same value we got from rtl_433. There is one more value which we need to measure, the gap period between the bursts.



For me this turned out to be 13ms.

Now that you have all the required information, it is time to generate the .ft file which rpitx can use as input. The .ft file format is used to easily implement digital modes. I doubt that you'll be able to get away with anything complex (my DSP knowledge isn't very extensive, I'm a programmer, not an engineer), but you'll be able to implement ASK and probably even FSK2 fairly easily.

The .ft file consists of 16 byte chunks. In theory it should be 12, an 8 byte float for frequency and another 4 byte block for duration. Somehow another 4 bytes have crept in there (maybe it is using 8 bytes for duration), so a chunk consists of an 8 byte block for frequency, a 4 byte block for duration and another four bytes which you can just fill with 0s.

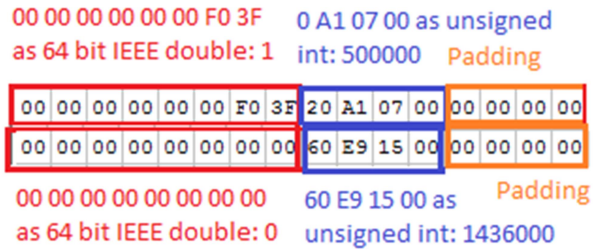
The first block is a frequency drift/offset value in Hz, encoded in the IEEE 754 floating-point "double format" bit layout. The next 4 bytes is a 32bit unsigned int, representing the duration which the signal has to remain on air, measured in nanoseconds. This is followed by 4 bytes of 0 padding.

Setting the frequency value to 0 tells rpitx to use an amplitude value of 0 (which is what we'll be using to generate the gap periods). When I generated the signal I just used a frequency offset value of 1 when generating a pulse, you don't have to use the actual frequency of the signal (434M030), you'll just tell rpitx to transmit at 434M030.

So to transmit the short pulse and its gap period, I send:

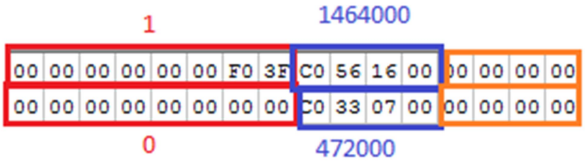
Frequency:1Hz	Duration:500us	Padding	Frequency:0Hz	Duration:1436us	Padding

In a hex editor, the 32 bytes looks like this



And the long pulse and its gap period

Frequency:1Hz	Duration:1464us	Padding	Frequency:0Hz	Duration:472us	Padding



You need to remember to insert the gap period between bursts, you cannot just transmit one continuous stream.

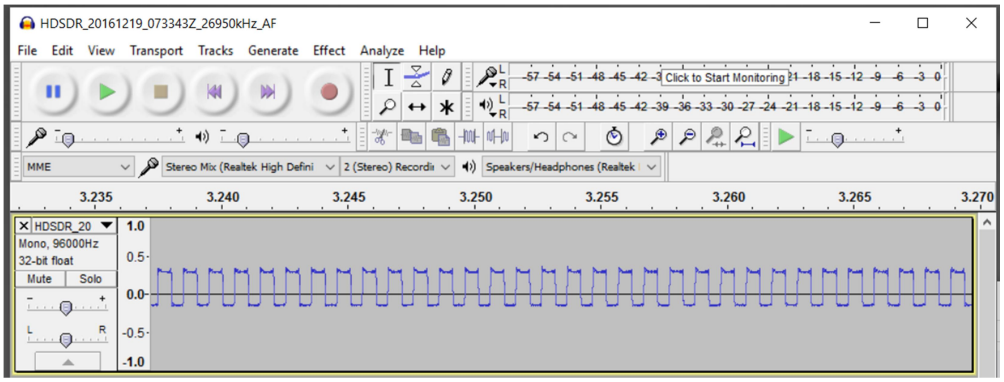
Once the .ft file has been generated (let's assume we called it doorbell.ft), it can be transmitted using rpitx using the following command

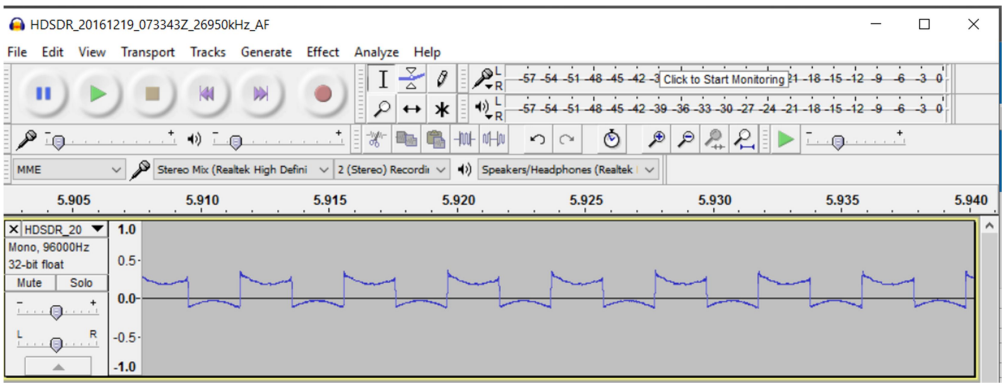
sudo rpitx –m RF –i doorbell.ft –f 434030

Note: It'll take some tweaking and playing around to get it just right. You might have to pad with silence (frequency = 0), before you start your actual transmission. You can use rtl_433 to check what bit sequences you are transmitting.

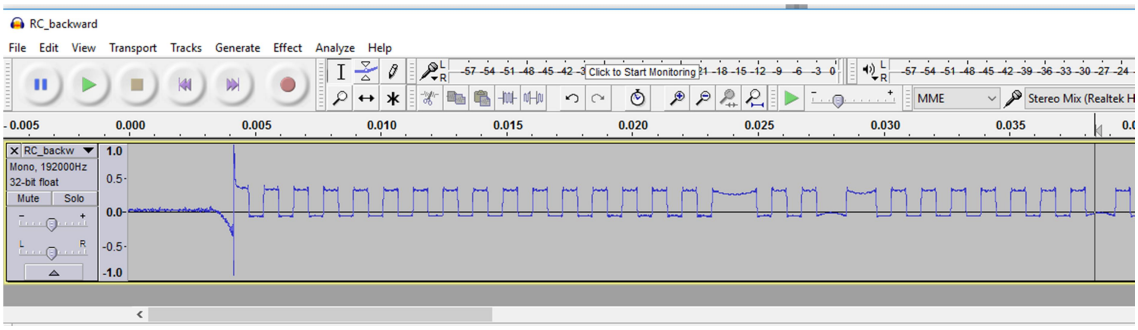
RC Cars

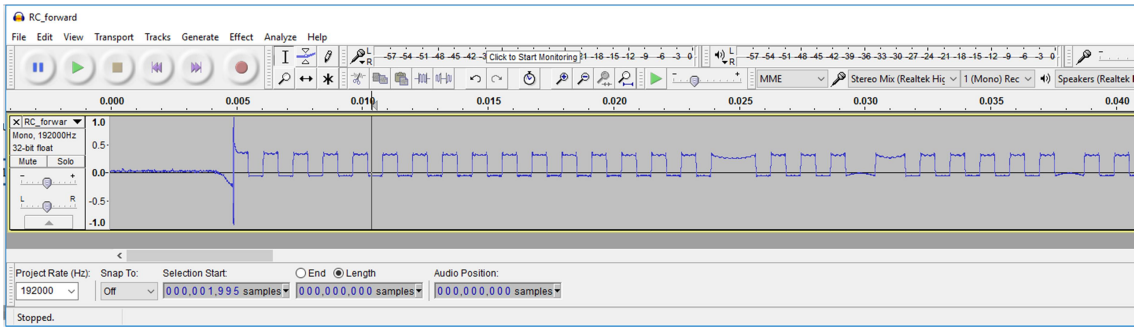
RC cars differ widely, so you'll need to go find the patterns for the car you are using. A really cheap car I was experimenting with at first (could only go forward or backwards) used simple OOK with alternating on and offs. The transmitter would just transmit with different symbol periods (or baud rates) depending on whether you wanted to go forward or backwards. The first picture below was for forward and the second for backward



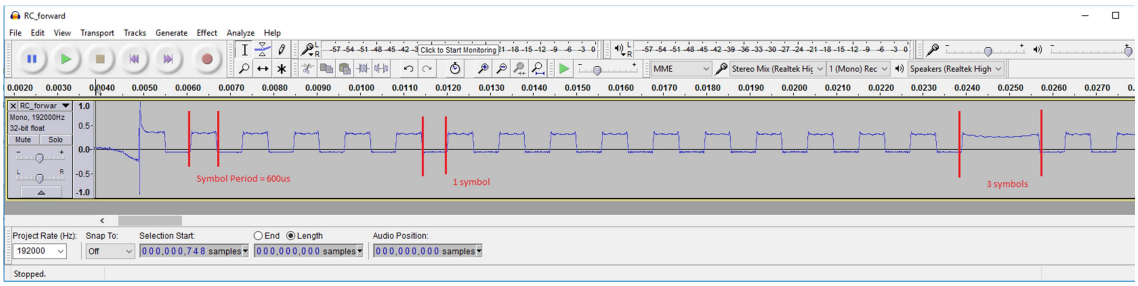


I bought a second, slightly more expensive RC car which could also turn, and it turned out that it had a much more complex way of transmitting commands. It also used OOK, but there were different patterns for each command.





What I found to be the easiest for OOK is to find the symbol period (just look for the shortest “up”) and then just map everything in terms of up and down. If you look at 2 pictures above, you’ll see that they both start with the same sync pattern: 16 pairs of up-down followed by 3 up symbols. You’ll just have to experiment with your car and see.



Once again you’ll have to generate the .ft file and transmit it via rpitx. You’ll have to go track down the frequency of your RC car, this is usually on the car somewhere (or on the packaging), but if it isn’t you can have a look around 27MHz and 40MHz.

There are various ASK and OOK devices. Hopefully by having gone through this exercise you will have gained some insight as to how to generate the signals which these devices use, and maybe you can find some other uses for it (you could try to spoof fake readings to a temperature sensor or open a gate or garage door). What you do with this is up to you. Enjoy!