Goal of the project:
The goal of the project was to receive wireless M-Bus packets with an RTL2832-based on DVB-T receiver. Wireless M-Bus Transmitters are hard to miss: water meters, heat cost allocators or smoke alarm devices. Most of them could be easily found in nearly every house.

Communication method:
In wireless M-Bus the following communication methods have been using according to [TDA5340_AN_WMBus_v1.0.pdf]:

Wireless MBUS Mode	RF Centre Frequency [MHz]	Channel Bandwidth [kHz]	TX frequency variation [+/-kHz]	Modulation	Min Deviation [kHz]	Max Deviation [kHz]	Data Rate [kchips/s]	Coding	Transmission duty cycle [%]	Bidirectional
S1	868,3	600	50	2-FSK	40	80	32,768	Manchester	0,02	No
S2	868,3	600	22	2-FSK	40	80	32,768	Manchester	1	Yes
T1 and T2 (METER)	868,95	500	50	2-FSK	40	80	100	3 out of 6	0,1	No
T2 (OTHER)	868,3	600	22	2-FSK	40	80	32,768	Manchester	1	Yes
R2 (OTHER)	868,33	60	17	2-FSK	4,8	7,2	4,8	Manchester	1	Yes
R2 (METER)	868,03 +n*0,06	60	17	2-FSK	4,8	7,2	4,8	Manchester	1	Yes
C1 / C2 (METER)	868,95	500	22	2-FSK	43	47	100	NRZ	0,1	No / Yes
C2 (OTHER)	869,525	250	22	2-GFSK (BT =	23	27	50	NRZ	10	Yes
F	433,82	1740	70	2-FSK	4,8	7	2,4 / 4,8	NRZ	10	Yes

Physic layer summary of the supported W-MBUS mode

After scanning for frequencies in the table above, some activities were discovered close to 868,95 MHz. That corresponds good with T1/T2 and C1/C2. T2/C2 stand for bidirectional and T1/C1 for unidirectional communication.

Sample rate selection:
Given with the table above, we seen the channel bandwidth of 500 kHz (for C1 and T1 modes). With the DVB-T receiver we will get a complex signal, thus the bandwidth is equal to a sample rate.

So, a setting sample rate of 500 kS/s would be sufficient. Unfortunately that is impossible, according to librtlsdr.c:

```
/* check if the rate is supported by the resampler */
if ((samp_rate <= 225000) || (samp_rate > 3200000) ||
   ((samp_rate > 300000) && (samp_rate <= 900000))) {
        fprintf(stderr, "Invalid sample rate: %u Hz\n", samp_rate);
        return -EINVAL;
}
```

Desired sample rate seems to be unsupported by the hardware! So oversampling must be used - sample rate of 1 MS/s is the next higher possible sample rate.

The signal processing flow
The signal processing done in the program is:

rtl_sdr -> i, q -> [pre-filtering, sample rate decimation] -> demodulation -> post-filtering -> clock recovery -> bit slicing -> packet decoding.

DVB-T receiver will be configured by “rtl_sdr" for desired sample rate and carrier frequency. You have to install rtl-sdr and libusb first - consult documentation of rtl-sdr-Project to do this properly. rtl-sdr throws iq-samples either in a file or on stdout.

2-FSK-Demodulation is computationally seen an intensive step, the amount of samples will be decreased (decimated): every seconds sample will be dropped off after preceding filtering.

Decimation requires the low-passing of the signal. If the sample rate is to be decreased by the factor *k*, the low-pass-filter is to be designed for a new sample *rate = old sample rate / k*. Without low-passing of the signal some troubles with aliasing are possible. As we will see later, carefully choosing filter parameters has enormous impact on packet receiving rate.

The input filter is designed as symmetrical low-pass FIR-Filter, because the phase of signal remains untouched. For the purpose of computational reduction polyphase filters were implemented altough.

FSK-Demodulator is a simple polar discriminator.

A so called Time-Square-Method has been used for clock recovery. Shortly, the square of signal passing bandpass with passband close to data rate will generate rectangle pulses accordingly to data rate. In the present case the data rate is 100kHz. The band pass filter is of Type 1 Chebyshev in order to maximize stop band attenuation and to reduce CPU utilization. A Butterworth-Filter would require more CPU power and was considered as unusable.

Bit-Slicing: The signal from demodulator will be “sampled” with reconstructed clock and put as logical 0 or 1 out.

Preamble detection: the bit stream from bit slicer will be searched for the packet preamble. That means the stream will be XORed with packet preamble and number of different bits would be counted. As soon as number matches threshold, the bit will be marked. GNU Radio has a similar block, called “Correlate Access Code”, that will do pretty the same.

Packet decoding: packet decoder searches for marked bits and does decoding. Checksums will be checked and, in case of T1-Mode, 3 out of 6 code will be mapped to “normal” data.

Preambles are given in the table below [TDA5340_AN_WMBus_v1.0.pdf]:

Mode	Preamble / Synch sequence (chips)	Postamble (chips)
Mode S1	n × (01) 0001110110 10010110 $n \geq 279$	n × (01) $1 \leq n <$ 4
Mode S2	n × (01) 0001110110 10010110 $n \geq 15$ or $n \geq 279$	n × (01) $1 \leq n <$ 4
Mode T1 and T2 (METER)	n × (01) 0000111101 $n \geq 19$.	n × (01) $1 \leq n <$ 4
Mode T2 (OTHER)	n * (01) 0001110110 10010110 $n \geq 15$.	n × (01) $1 \leq n <$ 4
Mode R2	n × (01) 0001110110 10010110 $n \geq 39$.	n × (01) $1 \leq n <$ 4
Mode C (transport layer A)	n x (01) 0101010000111101 0101010011001101 $n = 16$	
Mode C (transport layer B)	n x (01) 0101010000111101 0101010000111101 $n = 16$	
Mode F (transport layer A)	n × (01) 1111 0110 1000 1101 $n \geq 39$	
Mode F (transport layer B)	n × (01) 1111 0110 0111 0010 $n \geq 39$	

Preamble, Postamble and synch. sequence of selected wireless MBUS modes

Notes:
The experimenting shows, that receiving at higher sample rate (1.6MS/s) performs better. The sample rate was hard coded and cannot be changed. Filter have been redesigned with help of "Octave" accordingly the sample rate. See m-files in "filter" - directory. You have to choose between better receiving quality or computational intensiveness. Higher filter order has impact on receiver quality, lower filter order means fewer CPU utilization. Polyphase filters should be used to reduce CPU utilization, too. On FPU-less CPUS you can choose fixed point filter implementation, which is done only for FIR-filters. Extending IIR-Filter (band pass) with fixed point implementation is straight forward. Furthermore, a simple moving average filter does the filtering job barely worser than a FIR (moving average filter is a kind of FIR), but requires much less CPU power, so that receiving could be done on Raspberry B+, too!

The effect of better receiving through oversampling could be explained by:

- higher signal-to-noise-ratio through higher sample rate;
- wireless M-Bus transmitter are not so accurate to norm and occupy more bandwidth as specified;
- more bits makes clock recovering more precise.

Precise setting of carrier frequency was impossible. You can calibrate your DVB-T-Dongle with rtl_kalibrate and feed calibrating value to rtl_sdr. Or check the signal spectrum and then restart rtl_sdr with a new carrier frequency. For example, with my DVB-T- Receiver I had to choose carrier 50kHz under the standard of 868.95 MHz.

Works not very well with Raspberry Pi B+; best performance will be achieved by overclocking. In this case I have turned the clock up to 1000 MHz. The job could be done easily by Raspberry PI2 on two cores.

On detecting packet preamble you can specify the number of different bits, see for rtl_wmbus.c: const unsigned ACCESS_CODE_ERRORS = 1u;.

You can decide to write your own packet decoder, which would be probably helpful for CPUs with multiple cores - signal receiving/filtering and packet decoding could be done on different CPUs. See rtl_wmbus.c: OutFunction out_function = out_functions[1];

Simple packet collision detection was implemented. This works by observing received signal strength indicator computed as RSSI = sqrt(i*i + q*q). As soon as RSSI-value falling below threshold
(t1_c1_packet_decoder.h: #define PACKET_CAPTURE_THRESHOLD 5u), packet decoding is restarted in hope to receive ongoing packet. This slightly improves receiving in "noisy" areas with a lot of transmitters.

Wireless M-Bus uses a CRC-Algorithm for verifying datagram content. According to [TDA5340_AN_WMBus_v1.0.pdf] the polynom is 0x3d65. To identify other parameters of the algorithm, such as start value, xor-value for input or output, use [CRC REVENGE] as follow:

```
reveng -s -w16 -p 0x3d65
```

After identifying parameters use pycrc.py to generate the code and print resulting "crc_tab" out:

```
pycrc.py --std=c99 --xor-out=0xFFFF --poly=0x3d65 --width=16 --algorithm=tbl --generate h -o crc.h
pycrc.py --std=c99 --xor-out=0xFFFF --poly=0x3d65 --width=16 --algorithm=tbl --generate c-main -o crc.c
```

Links:

[TDA5340_AN_WMBus_v1.0.pdf]:
http://www.infineon.com/dgdl/TDA5340_AN_WMBus_v1.0.pdf?fileId=db3a304336797ff90136c14855cb7030

[CULFW]: http://culfw.de/culfw-1.61.tar.gz, see for 3outof6 decoding

[CRC REVENG]: http://reveng.sourceforge.net, see for how to identify a crc algorithm

[PYCRC]: https://pycrc.org, see for table driven crc algorithm implementation

[FHEM]: http://fhem.de/fhem.html, if interesting in how to interpret datagarams