µSDR–Pico: Next

Arjan te Marvelde, Version 4.1 – Nov 2024 (initial version uSDR-Pico: May 2021)

Since 2020, Raspberry offers a module based on their processor developed in-house: the RP2040. This processor contains a dual core, 125MHz Cortex based controller with plenty of Flash and RAM. This is a big step ahead, compared to the regular Atmel based Arduino modules. It also has many highly configurable and programmable I/O pins, which make integration of this module very easy for many applications. The C-SDK is mature by now and provides a quick start in actually setting this Pi-Pico to use.

This document describes the ***Next*** phase implementation of ***µSDR-Pico***, a small SDR implementation, inspired by Hans Summers' QCX and everything that followed after that, such as µSDX. The main deviation in hardware is the use of full FST3253 based mixers for both RX and TX paths, which makes modularization and experimentation a bit easier. The RX and TX front-ends, the control, signal-processing, audio i/f and mixer parts can in principle be built separately. Refer to the block schematic in section 4.

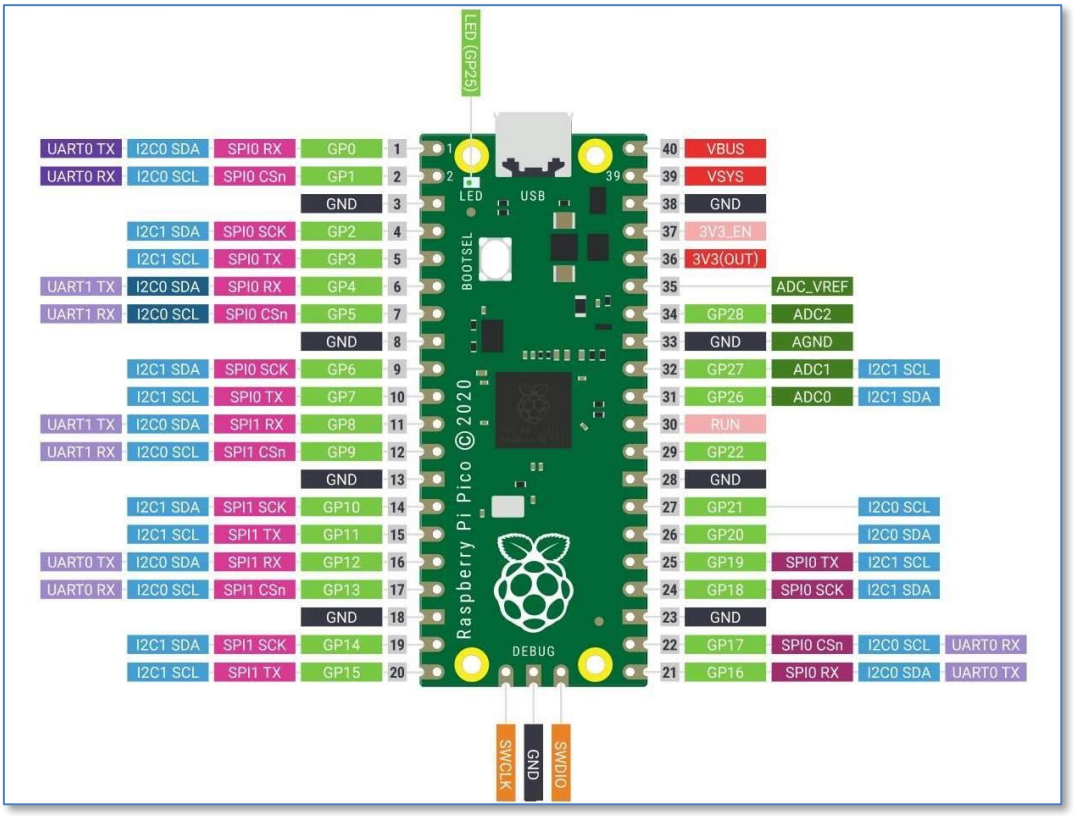
The image above shows the *µSDR-Pico* at date of writing, where the RX an TX parts are working and modules are built into a Teko enclosure.

Note to builders: This project is intended to be highly experimental and will always remain a work in progress, I try to spend some time on it every now and then but it is just a hobby. So, feel free to copy this project and add or change whatever you like. It is intended for experimentation and hence should not be considered as a flawlessly working kit.

The project is published on GitHUB: https://github.com/ArjanteMarvelde/uSDR-pico. That version has been tested, and is ready to be used as starting point for your own SDR experiments.

1 Raspberry Pi – Pico (RP2040)

The Raspberry Pi – Pico module is the core of this project, so below are the pinout and the way it is used in *µSDR-Pico*:



Raspberry Pi - Pico, pin usage for µSDR-Pico project					
Auxiliary GPIO		GP0	Vbus		Not used
Auxiliary GPIO		GP1	Vsys		5V power input
		GND	GND		
Auxiliary GPIO		GP2	3V3 en	x	Do not connect
Auxiliary GPIO		GP3	3V3 out		3V3 to peripherals
Encoder A input		GP4	Vref		ADC range, zener down to 3V or so
Encoder B input		GP5	GP28		ADC2: Audio input
		GND	GND		
Aux button 1 input		GP6	GP27		ADC1: QSD Q-channel input
Aux button 2 input		GP7	GP26		ADC0: QSD I-channel input
Aux button 3 input		GP8	RUN		Pushbutton to ground for reset
Aux button 4 input		GP9	GP22		PWM 3A: Audio output
		GND	GND		
Not used	x	GP10	GP21		PWM 2B: QSE Q-channel output
Backlight to LCD		GP11	GP20		PWM 2A: QSE: I-channel output
Reset to LCD		GP12	GP19		PTT control output
Data/Control to LCD		GP13	GP18		PTT switch input
		GND	GND		
SPI1-SCK to LCD		GP14	GP17		I2C0 SCL: Si5351A, BPF, RX
SPI1-Tx to LCD		GP15	GP16		I2C0 SDA: Si5351A, BPF, RX

2 Principle of operation

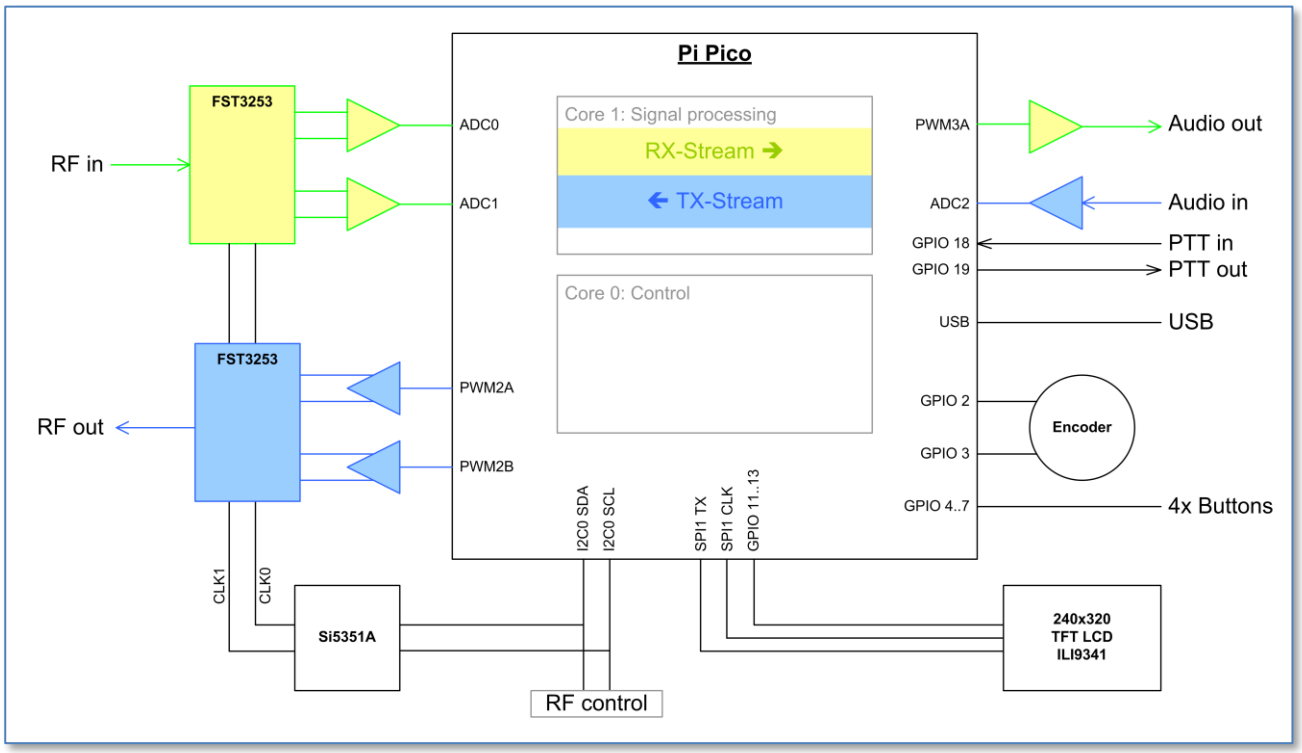


Figure 2-1 µSDR-Pico principle of operation

The block diagram shows the processor board and several peripherals. The VFO is based on a Si5351, in this project implemented with an Adafruit board. The VFO clocks the receiver (Quadrature Sampling Detector, QSD) and transmitter (Quadrature Sampling Exciter, QSE) mixers, which are based on the FST3253. In principle, any other mixing stage can be used that produces a quadrature RX signal and consumes a quadrature TX signal. RF control (band filter switching, attenuator/LNA selection, VFO settings) is implemented over the I^2C bus.

Note that the ADCs (and DACs) have a maximum range of 3V3 which in case of ADCs is zenered down to about 3V in order to suppress noise. Therefore the analogue input signals should be limited accordingly.

On the user side, there are interfaces for Audio I/O, PTT switch, a Rotary Encoder, 4 Buttons, a SPI interface interface to control LCD and 4 auxiliary GPIOs.

In addition the Pico USB interface is made available externally, which can be used for device programming and also as stdio based debug monitor replacing the UART in previous versions. Note that the UART could still be used since the pins are two of the four free GPIOs made available as auxiliary interfaces.

The display has been changed from an alphanumeric to a graphical one, since that gives much more freedom in choosing an appropriate user interface layout.

The new hardware implementation is also optimized, since the VFO is moved to the mixer board and the audio interface circuits are included on the CPU board. As a consequence, the boards will become slightly larger. There will also be a better bus interface, for power and control signals. This allows for a proper 5-board stack with less ad-hoc wiring. See section 4 for the details.

The Pico itself contains two cores, of which core0 is used to do all non-realtime processing such as user interface and othe rcontrol stuff. The other core1 is dedicated to the realtime signal processing and follows a fairly strict timing scheme. To ensure operation times, most software in this core1 is pre-loaded into RAM.

3 Software

Summary:

The processor has two cores that work to a large extent independently, apart from some hit and miss waits caused by memory access arbitration. The idea is to let ***core1*** do all the realtime signal processing, while ***core0*** (the default at startup time) does all the less time critical control stuff.

> ***Note*** that the Pico uses an eXecute In Place (XIP) Flash interface, meaning that code is really executed from a relatively small cache-memory section of the SRAM. Normally this is okay, but time-critical parts of the code can also be loaded in SRAM at boot time; the Pico has plenty memory. Also, the time critical stuff is located in ***core1***, and therefore this core is given priority over ***core0*** in case of arbitration for access to shared resources.

The signal processing on ***core1*** is split up in two streams, an RX and a TX stream. All three ADC inputs are sampled on highest speed in Round-Robin fashion, and the interrupt handler merely copies the most recent conversion results into buffers when the ADC FIFO fills up. This way, ADC samples for every channel are taken during a period of 6 µsec somewhere within the sampling rythm.

The RX stream takes the I and Q samples from the QSD, processes these and outputs samples through a PWM-based DAC to the audio interface. The TX stream does the reverse, taking the samples from the audio input, process them and sending PWM-DAC I and Q signals to the QSE.

For the processing of the samples, one out of two mechanisms can be selected during compile time; the first processes everything in the time domain while the second processes in the frequency domain. The time-domain mechanism processes the signal sample-by-sample, and therefore has a very short loop. The frequency domain mechanism collects samples in a buffer which is converted to frequency domain by means of an FFT. After processing the reverse is done with an iFFT.

The time domain processing runs on a 64µsec basis (15.625 kHz) and the frequency domain processing every 32.768msec (i.e. 512 x 6 µsec). In both cases, the ***core1*** timer callback function takes care of the sample transfer and triggers the DSP loop at the right moment. The DSP main loop processes the actual RX and TX streams.

The Pico has two I^2C buses, but only one is used to control the VFO and the relays. The display now is a larger graphical than the alphanumeric of previous versions, and this is connected through a faster SPI bus. There are four GPIOs reserved for auxiliary buttons. These are currently used as direct controls, but this could easily be upgraded with a binary decoder to increase the number of control lines. Two more GPIOs are used to connect a rotary encoder and another two as PTT in and output.

Files overview:

uSDR.c	The main loop and system initialization. The main loop runs as a separate process on *core0* and takes care of all user interfacing.
dsp.c	Signal processing loop, handles the sampling and invokes TX and RX branches. This loop runs as a separate process on *core1*.
dsp_tim.c	Time domain signal processing engine
dsp_fft.c	Frequency domain signal processing engine
fix_fft.c	Fixed-point Fast Fourier Transform functions
si5351.c	Control of the VFO module, that provides I/Q clocks to the QSD and QSE.
lcd.c	LCD output support and 16x2 byte output buffer.
hmi.c	The user interaction, handling the control events.
monitor.c	A command shell running on the stdio UART.
relay.c	Controls the various relays through I^2C expanders.

3.1 Main (uSDR.c)

The file `uSDR.c` contains the main loop and startup sequeces. All other C modules have their own local initialization routine, which are invoked sequentially after initialization of some generic interfaces. The signal processing on ***core1*** for example, contained in `dsp.c`, is started from `dsp_init()`.

The main loop itself is triggered by a callback routine hooked to a repetitive timer with a timeout of about 100msec. It mainly handles the functions that have to do with the user interface, so handling the buttons and encoder, refreshing the display and also updating the VFO settings. Also the command line interface monitor is timed in this loop, since it is a user interface too.

So ***core0*** contains the control and HMI related software, which is not hard realtime. The ***core1*** on the other hand contains the signal processing chains and requires close attention to make sure that the processes are finished in time. Note that the Pico can be overclocked to create some more processing room, but this may result in undefined behaviour and possibly additional noise. Still it is worth to experiment with this.

3.2 Signal processing

Where the other parts are enabling- or control functions, the signal processing forms the real heart of the uSDR-Pico. It is what this project is really about. Read about it in this subsection.

3.2.1 Sampling and timing (dsp.c)

The structure is built in such a way, that the time- or frequency domain signal processing can be selected at compile time by defining a directive in `dsp.h`. The corresponding c-sources are then included integrally in `dsp.c`.

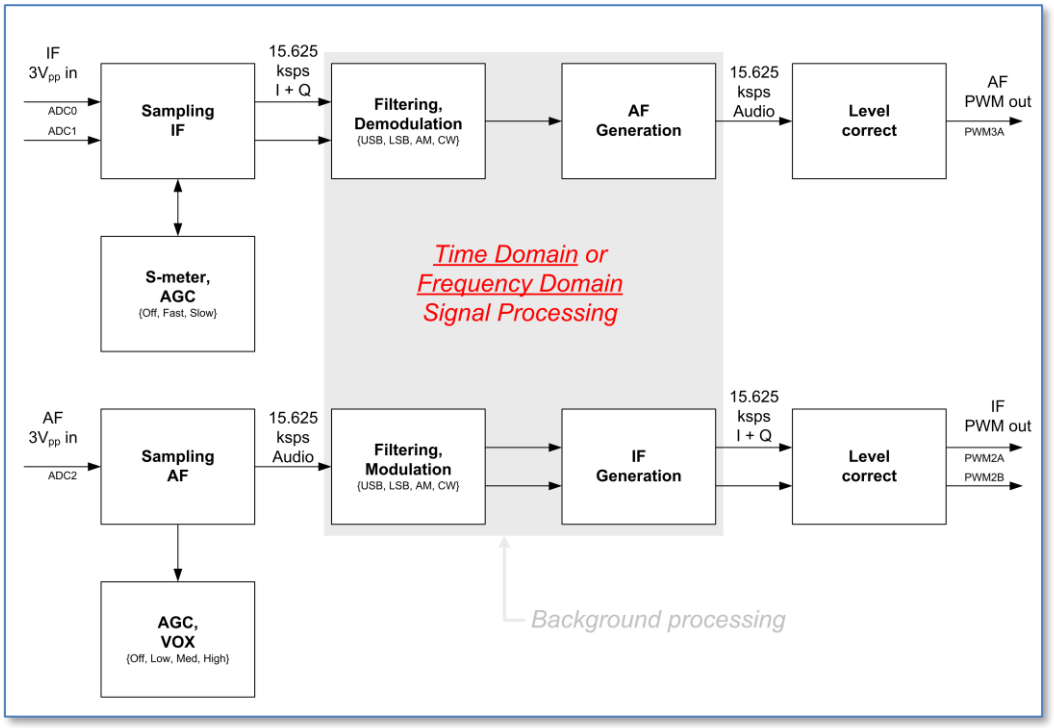


Figure 3-1 Timer callbak and background signal processing on core1

The main part of `dsp.c` contains two functions that are allowed to run at a different rate. One function is a timeslot timer callback routine, which runs at sampling rate every 64µs (i.e. timeslot) and takes care of preprocessing like level detection for AGC/VOX and the timing. The other function is the main loop that is triggered by the timeslot routine to acquire and prepare the preprocessed samples and do the actual SDR signal processing in the background.

The timing obviously works different in both methods; in the time-domain (TD) case the RX/TX streams need to be invoked for each sample (timeslot) and in the frequency domain (FD) case this is only every time an FFT buffer is filled with samples.

The sampling process uses the ADCs in free-running round robin fashion for all three channels, while a DMA process stores the samples in dedicated queues until full. Each ADC conversion takes 2µsec and after 3 x Queuedepth conversions the ADC and DMA tandem will block.

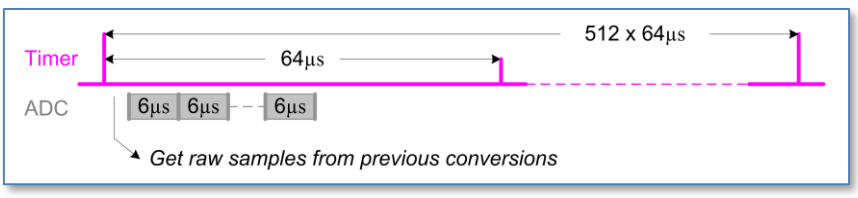


Figure 3-2 TD and FD timing of background signal processing

Figure 3-2 shows the timing of the sampling and the timer callback timeslot routine. After a timeout, a new 64µs slot starts with purging the sample queues and starting a new sampling burst. This timing rythm is key in the whole processing chain and effectively results in a 15.625 ksps sample rate. Each integration cycle can add up to 10 samples per channel (i.e. 10x 6µsec) for increased dynamic range and somewhat averaging out ADC errors.

Sample preprocessing is done every timeslot inside the timer callback routine, but both TD and FD processing is executed as a background process, in the dsp-loop, which is triggered after the preprocessing is done. TD processing is triggered at every timeslot, while frequency domain processing is only triggered every 512 timeslot interrupts, when a ½ FFT buffer is filled. The TD or FD dsp-loop signal processing must be finished before the next sample or buffer is available, or overrun will occur. For TD processing it means that the remainder of the 64µs timeslot is left to use. For the FD processing this remainder can be multiplied with half the FFT buffer size, i.e. 512.

HMI and other control or I/O functionality runs on the other core, so will not interfere as long as no common resources are used.

3.2.1.1 Sample preprocessing

The timer callback routine determines the sampling rate actually used in signal processing, and this is set to 64 µsec (15.625 kHz). ADC input samples are more or less continuously streamed to buffers of length up to 10, by means of DMA.

The phase error between the I- and Q-samples is the duration of one ADC conversion; 2 µsec. This results in a phase difference of about 1% in the audio domain (at a frequency of 4kHz), and hence there is no real need for intermittent sampling and the required phase correction by averaging the last two samples on the I channel. Such an averaging process in fact would result in a much larger distortion.

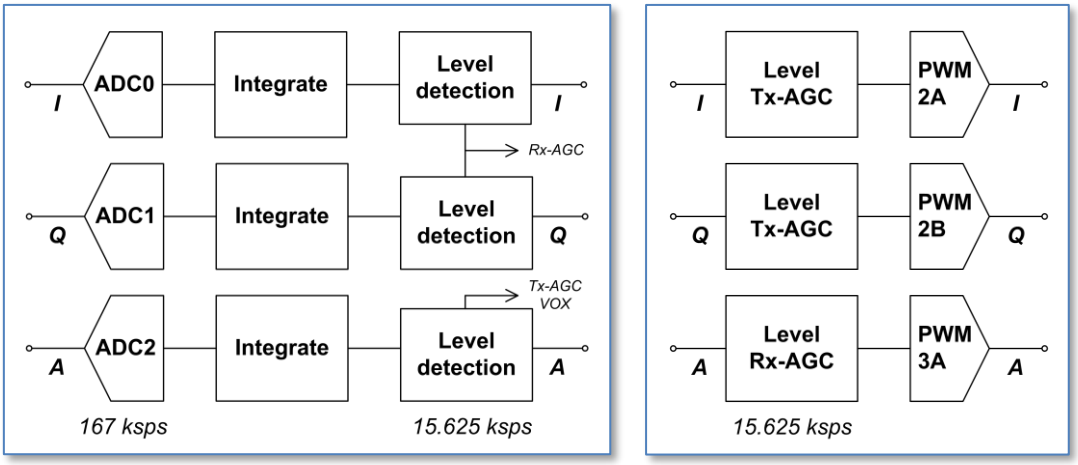


Figure 3-3 Timer callback routine processing overview

In Figure 3-3 both input and output streams are shown for IF and audio. Although these can technically not be simultaneously active in half duplex mode, the I/O parts shown here are still executed every timeslot. The background signal processing is executed either in RX or TX direction.

Integration and DC-bias removal

The samples collected in the sample queues are integrated for the 3 input channels (Q, I, A). The resulting values from the ADC are electrically centered on approximately half the reference voltage, set by the resistor bias circuit, which also corresponds with half the ADC dynamic range (2048). This level needs to be determined precisely and subtracted from the actual samples before further processing.

The DC-bias estimation is based on a running average low-pass filter:

```
bias = β*sample + (1-β)*bias
```

where the fraction β and sample rate determine the low-pass RC time, and lower values of β yield lower corner frequencies.

$$RC\text{-}time = sample\text{-}time * (1-\beta)/\beta$$

A fast fixed point algorithm can be implemented when powers of 2 are used:

$$bias = sample/2^n + (2^n - 1)*bias/2^n = bias + (sample - bias)/2^n$$

$$bias \mathrel{+}= (sample - bias)/2^n$$

This holds the risk of underflow, (sample – bias) needs to be larger than 2^n to be significant. This can be avoided by left shifting the values as in proper fixed-point processing.

The sample time at this stage is still 6µs, so the RC time of this 1st order low pass IIR filter is: $(2^n-1)*6\ \mu s \approx 12.3msec$ or 81Hz when n = 11.

A different method uses a moving average to determine DC level. This method is easier and has less scaling issues. The samples of a history N deep are summed, the average is obtained be simply dividing the sum by N. Depending or variability in the sample stream, N must be chosen higher or lower; the passband is approximately $f_s/2N$.

A delay line of sample values needs to be stored, of which the oldest is subtracted from the averaging sum after the latest new sample was added to it. Then the value in the delay line is exchanged with that new sample value and the pointer advanced to the next.

Again, the dividision can be speeded up by chosing a power of 2 for N and right shifting instead of dividing.

I use one new sample every timeslot (64µsec) and a delay line of 256, which yields a low pass f_c of about 30Hz.

Level detection

Two level detectors are also part of the preprocessing function, one maintaining receiver level (RSSI) and the other maintaining the audio level (for VOX). The RSSI is determined from the (I, Q) vector length and the VOX level from the audio sample stream.

The RSSI is determined by means of an α-max + β-min approximation, to avoid time consuming sqrt approximation. The deviation with the chosen coefficients is less than 2.4%, which is good enough for the purpose.

The RSSI and VOX levels are low-pass filtered by means of IIR, the RC time is adjustable and set to 16msec (65Hz).

The RSSI is input for the S-meter value function. The scale for this is logarithmic: from S1 up to S9 each level corresponds with a 3dB RSSI step. Absolute S9 signal level at the antenna is defined to be -73dBm in 50 Ω, which corresponds with 50.1µV. Assuming unity gain in the mixer, the total voltage gain is primarily determined by the IF amplifier, which is set to approximately 300x. Then S9 should correspond to roughly 14.4mV signal on the IF output/ADC input. Every +10 step above S9 multiplies this voltage level with $10^{0.5}$=3.16.

S-meter	Mixer input [µV]	IF output [mV]	RSSI level (relative)
9+40	5006	1500	25600
9+30	1583	469	8096
9+20	500.6	147	2560
9+10	158.3	46.0	810
9	50.1	14.4	256
8	25.1	7.19	128
7	12.6	3.59	64
6	6.30	1.79	32
5	3.16	0.90	16
4	1.58	0.46	8
3	0.79	0.23	4
2	0.40	0.11	2
1	0.20	0.06	1

In the signal acquisition there is an integration gain factor, adding 8 samples stretches the effective ADC range, which theoretically limits the RX sensitivity at the mixer input to about S2. In practise the RF noise floor will be much higher than that though and also the RF gain/attenuation needs to be taken into account. The S-meter should be calibrated at S9 level RF signal input with with no RF gain or attenuation active.

An AGC factor can be derived from the RSSI, which should be used to control the RF/IF amplifiers. For now this has not been implemented. A second use of AGC is to suppress fading, which can be done by controlling the AF output signal. For this a proper AGC mechanism should be implemented with adjustable attack and decay times.

VOX

The voice activated switch (VOX) algorithm needs to continuously test the Audio ADC level, so this is performed both in RX and TX mode. The VOX takes the Audio stream level and takes it through a low-pass filter. The VOX linger time determines how long it will remain active after the level dropped below the VOX threshold.

The actual audio level is determined in the timeslot timer callback routine, but the VOX algorithm is implemented in the background dsp-loop.

Sample buffering

The buffering of samples differs between TD or FD processing. In the TD case, every incoming sample is treated by the signal processing function individually, so a one-sample buffer is sufficient. Same applies to outgoing samples.

For FD signal processing incoming samples are stored in the next half FFT buffer, while outgoing samples are copied from a similar half iFFT buffer.

3.2.1.2 Sample output

The output of samples to either the audio interface in RX mode or the QSE in TX mode is handled at the end of the timeslot timer callback routine, just before the dsp-loop is triggered. In case of TD processing the routine just transfers the latest calculated sample to the Audio DAC. The signal level is corrected with an AGC multiplier, range checked and either clipped or attenuated before sending it to the DAC.

For the FD based signal processing, the output sample is copied from the iFFT output buffer.

3.2.1.3 Invoking the signal processing

For the TD processing the effective sample rate is also the rate for the RX/TX signal processing routines. For the FD processing the RX/TX functions are invoked only every half FFT-size (1024/2) samples, i.e. when a buffer is filled, and run in background while sample collection for the next buffer goes on in foreground. This yields a much lower call rate of about 305Hz (1/32.768msec), but this obviously is compensated by the larger processing load required by the Fourier transformations.

3.2.2 Time domain signal processing (dsp_tim.c)

Note: I don't use this mode anymore, so it is still in for reference but in a future version it will be removed.

3.2.2.1 RX stream

The RX stream can be functionally divided into a part that does the actual demodulation type of choice, and a part that does the audio generation.

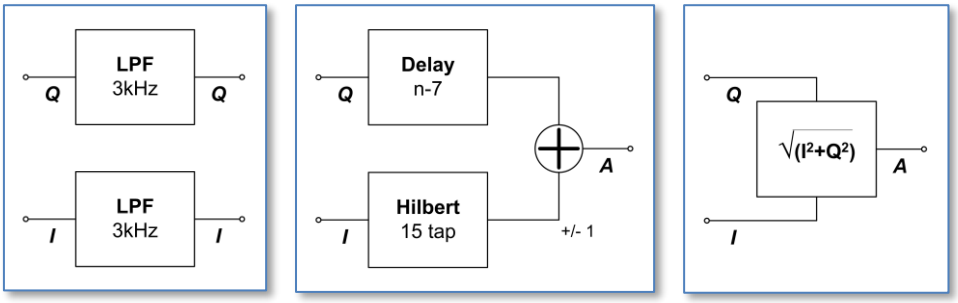


Figure 3-4 RX TD signal processing

Low pass filtering

First stage of the TD signal processing is low pass filtering of the signal. The only filter currently implemented is 3kHz, which is okay for SSB, but maybe too narrow for AM and too wide for CW. The Nyquist frequency is about 7500 Hz, so the maximum filter bandwidth should be lower than that.

Demodulation

For SSB demodulation, the incoming I and Q samples are stored in a 15-sample delay line. A 15-tap Hilbert transform on the Q channel is done and the result is subtracted from or added to the n-7 sample in the I delay line to obtain the USB or LSB audio output respectively.

For AM demodulation the length of the I-Q vector needs to be calculated, and no further transform is required.

The resulting audio sample is stored, to be used by the timeslot timer callback routine for output to the DAC.

3.2.2.2 TX stream

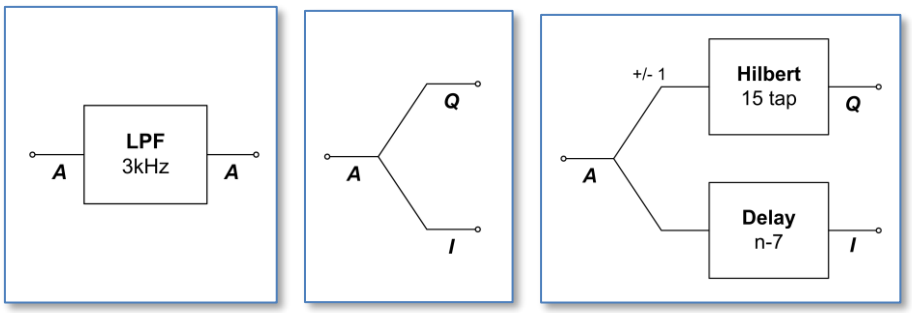


Figure 3-5 TX TD signal processing

The TX stream is the inverse of the RX stream: the same components can be found here as in the RX stream. As in the RX the SSB is generated by converting the I-samples into Q-samples through a 15 tap Hilbert transform. The Q output is multiplied by -1 for USB and +1 for LSB, the I-samples need to be delayed by 7.

The dynamic range of the sample streams is matched with that of the DAC range, before output, in the callback routine.

3.2.2.3 TD Filter implementations

Two TD signal processing blocks are of interest, the Hilbert transformation and the low pass filter. These are implemented as described in this section.

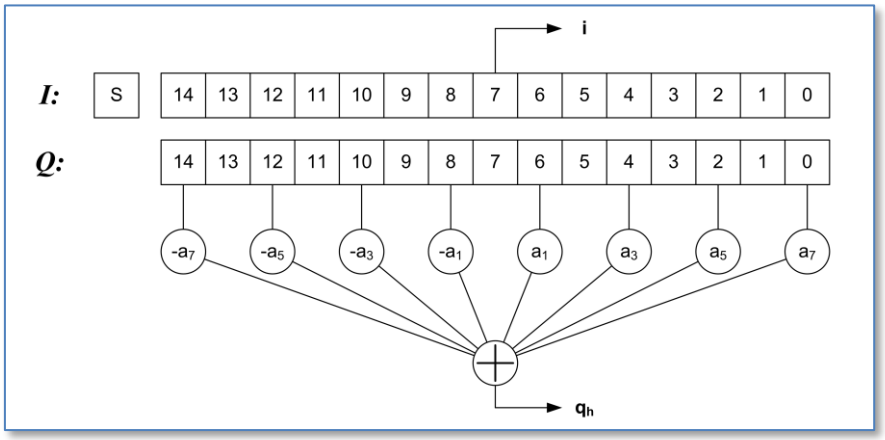


Figure 3-6 TD Hilbert transform

The Q delay line array has a length of 15, to enable a 15-tap classic Hilbert transform. The even samples have a zero coefficient so, as in the above figure, only 8 of the samples are used in the calculation. Due to symmetry of this classic

Hilbert transform only 4 multiplications have to be performed. The resulting transformed output is in phase with the 8th sample in the array, being I[7], Q[7] and the calculated Q_h.

The coefficients for the taps can be derived from the Hilbert transform rules combined with the choice of a proper windowing function. The window function suppresses the ripple otherwise seen in the frequency response. See for example Iowa Hills tools to obtain a set of coefficients.

The coefficients are given by:

$$h(n) = w(n) \cdot \frac{2}{\pi \cdot n} \qquad \text{where n is odd [-7, 7]}$$

In this function $w(n)$ is a windowing function, for example a Hamming (raised cosine) window:

$$w(n) = 0.54 + 0.46 \cdot cos\left(\frac{\pi \cdot n}{7}\right) \qquad \text{again, n is odd [-7, 7]}$$

Note that the bandwidth of the classic Hilbert transform is half the sampling frequency. The response at the edges drops off, so to get a response that extends far enough towards the edges, either the sampling rate must be lowered or the number of taps must be increased. For 15 taps the rate would be ½ the 64usec, appr. 7800Hz.

-7	-5	-3	-1	1	3	5	7
-0.00728	-0.03224	-0.13631	-0.60762	0.60762	0.13631	0.03224	0.00728

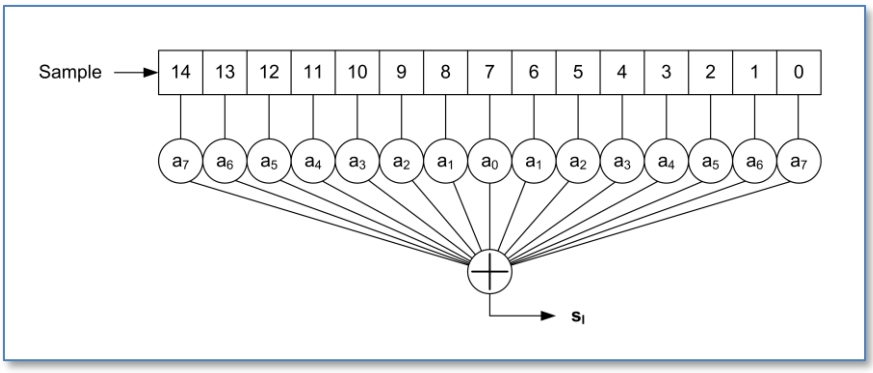


Figure 3-7 TD Low pass FIR filter

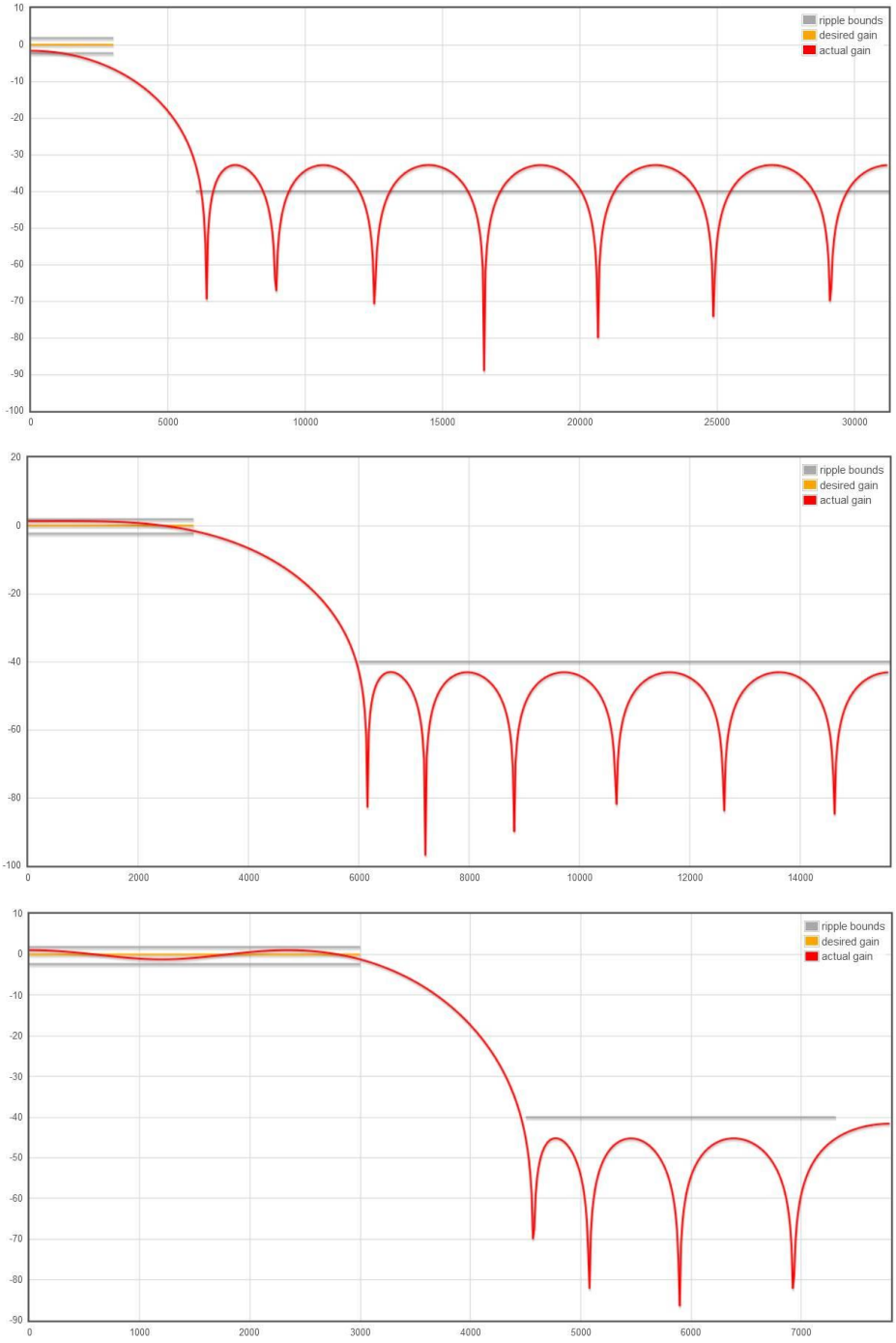
Several 15-tap low pass filters have been created, with a corner frequency of F_c=3kHz. The stop-band depends on the actual sample rate the filter is designed for. It usually starts around 5kHz, with a level of -40dB or better. The code contains filters for 62.5 kHz, 31.25 kHz and 15.625 kHz sample rates, although only first and last are actually used.

These low pass filters are simple symmetric FIR filters, that represent the impulse response of the desired low pass behavior. They consist of 15 signed integer arrays. Hence, per sample 15 multiplications and additions need to be done, but the RP2040 has a single cycle 32bit MPY instruction, so that be fast enough.

To find the proper coefficients, see for example Iowa Hills DSP tools or the T-Filter on-line calculator:

- http://www.iowahills.com/
- http://t-filter.engineerjs.com/

T-filter parameters 62500, 31250 and 15625 sample rates, passband 3kHz ripple <5dB, stopband from 6kHz at -40dB:



Note that for the 3kHz low pass filter, a 15 tap FIR algorithm works best at lower sample rates. For high sample rates it would be better to use more taps. Currently only the low sample rate filter is used in uSDR-Pico.

3.2.3 Frequency domain signal processing (dsp_fft.c)

While Time Domain signal processing is done on a sample by sample basis, the FFT requires a buffer full of samples and hence the signal processing is invoked every time a buffer is filled. Therefore, the Signal Processor is triggered only every 512 samples (512x64usec ≈ 32msec), and runs in the background while interrupted by the timer calback routine at raw sampling rate. See also Annex A for a more detailed discussion.

3.2.3.1 Buffer handling

The buffer structure is built up from ½ FFT-size buffers, doubled for the complex side of processing. Buffer overlapping is used to ensure a smooth glueing of the chopped-up sample streams. The handling is done inside the timer callback routine.

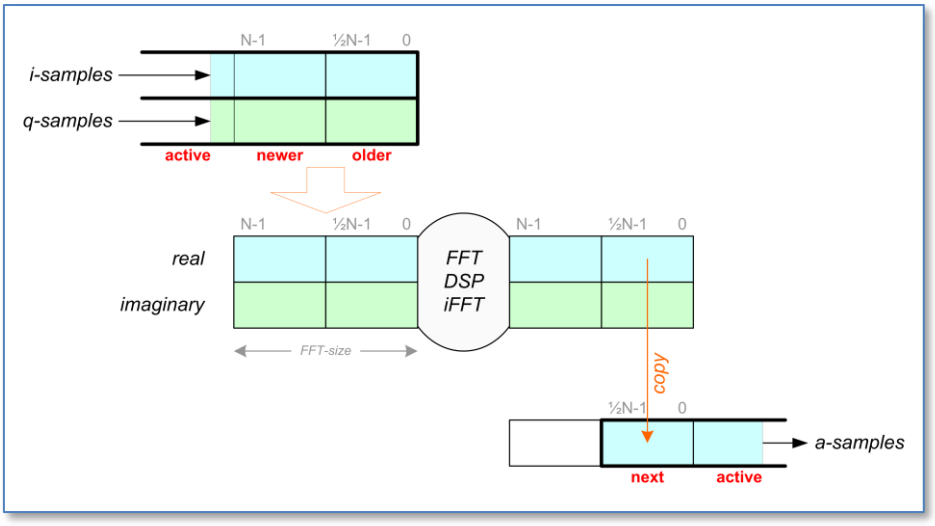


Figure 3-8 RX FD signal processing structure

The figure represents the RX case, the allocated buffers are in fact re-used for the TX case but work in opposite direction. The active interface buffer is one of a 3-buffer queue, the other being the saved samples of previous interval. The active buffers collect the I and Q samples captured by the timer callback routine. Whenever the active buffer is full, the input and output buffers are reorganized and the DSP loop is signaled to start.

The real part of the previous signal processing cycle result is copied to the output buffers. Then the saved input buffers are copied into the lower half of the FFT buffers, while the upper halves are zero padded. Then the signal processing cycle is begun, yielding a new result.

3.2.3.2 Signal processing

The signal processing engine follows the sequence of transformation to the frequency domain (FFT), applying the band filtering and shifting as well as transformation back to the time domain (iFFT). The samples in the audio domain are real, so a conversion from (and to) the complex representation is required before the FFT can be done.

RX case:

To avoid propagation of mixer artefacts around DC level, the target carrier frequency must not be directly downconverted to 0 Hz, but rather to somewhere in the center of the frequency band resulting from the FFT. This offset has to be accounted for in the used VFO frequency. With a sampling rate F_s=15.625kHz, and a Nyqvist frequency half of this, the *offset* frequency F_c should be somewhere around ¼·F_s ≈ 3.9kHz.

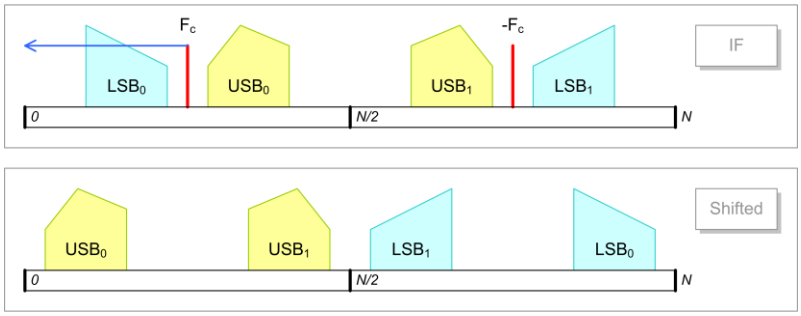
Just before handing over a sample buffer to the FFT and DS processing, the signal is shifted down to DC, by complex multiplication of the samples with a complex negative frequency sinusoid signal: $e^{(-j2\pi Ft)}$. Since the sub-carrier frequency is a quarter of the sample frequency, the complex factors for sample multiplication will be:

$[\cos(k\cdot\pi/2),-\sin(k\cdot\pi/2)] = (1,0);\ (0,-1);\ (-1,0);\ (0,1);\ \ldots$

The complex product series then results into a mere re-arrangement of the samples (I_k,Q_k) as follows:

$(I_0,Q_0);\ (Q_1,-I_1);\ (-I_2,-Q_2);\ (-Q_3,I_3);\ \ldots$

A graphic representation of this F_c shift:



Depending on the desired modulation mode, different sideband filters are applied to the spectrum buffer that is obtained after FFT. The filter merely removes the undesired parts of the spectrum, applying a raised-cosine rather than a rectangular window.

Again, a graphical representation of the result of this process (only real part of spectum shown):

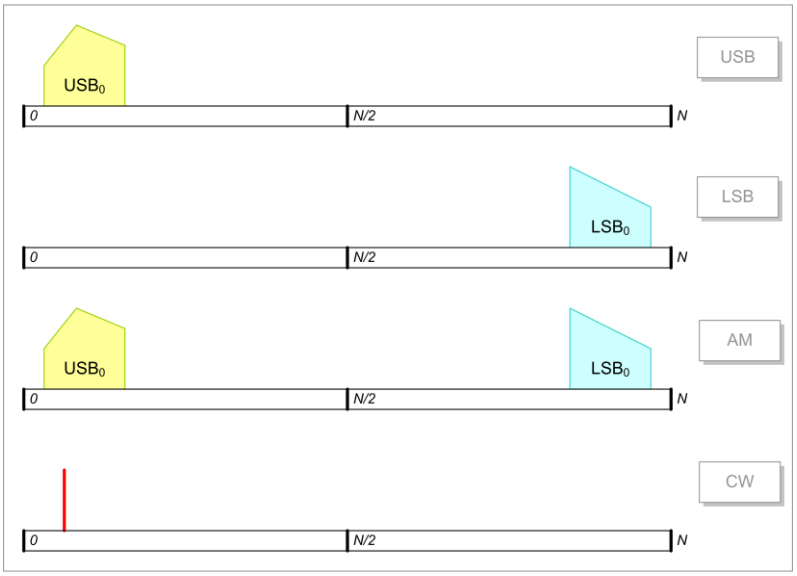


Figure 3-9 Spectrum and demodulation filtering

For CW the filter can be narrow around for example 900Hz so the CW signal is obtained when tuning to F_c – 900Hz. The frequency shown on the display should be corrected for this tone offset when in CW mode.

After the iFFT of this filtered spectrum, the real part of the complex time samples are copied to the audio samples buffer. For smooth time domain output the overlap-add method is used with half FFT-size buffers..

TX case:

The reverse actions are performed for transmission. The audio samples are copied in the real part of the FFT buffer, while the imaginary part is set to 0. Then after performing the FFT shift the spectrum up with the offset, filter out the desired spectrum and do the iFFT. Both real and imaginary parts are copied to the I and Q buffers.

Notes:

When shifting the spectrum, in fact a rotation is done; bins that shift beyond the FFT-buffer edge will re-enter on the other side.

When adding a carrier to obtain an AM baseband signal, this carrier should contain twice (?) the amplitude of any sideband signal

3.2.4 The Fast Fourier Transform (fix_fft.c)

Key element of the frequency domain signal processing is the application of the Fast Fourier Transform (FFT). This procedure transforms the time samples into frequency bins and vice versa with the inverse FFT.

3.2.4.1 Physical meaning of the FFT

An RF signal is mixed down with a direct conversion quadrature mixer, resulting in an in-phase (I) and a quadrature (Q) baseband, centered on DC. So what does this mean, for example to have negative frequencies? If you consider the original RF signal having two sidebands from amplitude modulation, the sidebands are just a bit higher or lower than the carrier frequency. When you mix this signal down with the carrier frequency, the lower sideband as a mathematical consequence is represented by a negative frequency.

Suppose that the mixed down signal is represented by the time dependent vector (I_t,Q_t). The rotation speed of the vector represents the frequency (DC doesn't rotate at all), the rotation direction represents whether the frequency is positive or negative. The actual movement of the vector is erratical, and contains a superposition of frequencies. With the fourier transform we actually want to analyze this set of rotation speeds (frequencies) and determine to what extent these are present in the (I_t,Q_t) sample stream.

To this purpose, the I and Q streams are converted in discrete (I_k,Q_k) sample pairs, which are the time-domain complex input to the FFT. At regular moments in time the latest N samples (i.e. the FFT size) are transformed into a frequency spectrum. This transformation period has to be shorter than what is represented by N samples, so there is some overlap.

2N *real* time samples would result in N *complex* frequencies (cf. Nyquist), where the missing negative complex frequencies are just a mirror of the N positive frequency bins. In contrast, 2N *complex* time samples result in N positive + N negative *complex* frequencies.

The bin with index 0 represents the DC component, and the bin with index N-1 represents the Nyquist frequency. The bins N and beyond represent the negative frequencies and bin 2N would be DC again. For representation, rotating the set of frequency bins with N will place the DC component at bin N and the upper/lower sidebands on either side.

Demodulation of SSB would boil down to filtering away everything but the 3kHz or so on the high side of the DC bin before transforming back to the time domain. Since upper and lower sidebands are different, it is clear that complex time samples are required to obtain the right transformation.

To get rid of noise around DC, the QSD mixing frequency could be chosen lower than the actual RF carrier, causing the baseband signal to land somewhere in the middle of the positive frequency bins, i.e. around N/2.Then after filtering, the baseband can be shifted down by N/2 before applying the inverse FFT.

3.2.4.2 FFT implementation

The FFT algorithm is explained in more detail in an appendix. The *uSDR-Pico* implements a very ligtweight version of the transformation.

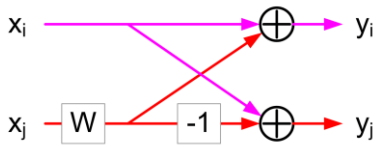
The first stage of the FFT algorithm is the reordering of the time domain samples; to be precise, the samples with indexes that are bit-reverse of each other need to be swapped. The array size is 1024 samples, so the index is 10 bits. The swap is then for example between samples [1111000001_b] and [1000001111_b]. This re-ordering is done before the FFT is calculated, and therefore named Decimation in Time (DIT), as opposed to the post-FFT DIF. The re-ordering is needed to enable an efficient chain of butterfly executions.

The point of the algorithm is how it goes through the array only once, i.e. avoid swapping samples back again. A general approach would be:

```
for (i=0; i<1024; i++)
{
    j = bit_reverse(i);
    if (i < j)
        swap(data[i], data[j]);
}
```

but the `bit_reverse` routine could take quite some time. A faster alternative (utilizing the relatively abundant RAM) is to use a lookup table instead. After that, the swapping of the samples is straightforward.

The second stage consists of a number of nested loops, calculating and adding the butterflies. One such butterfly can be represented as follows:

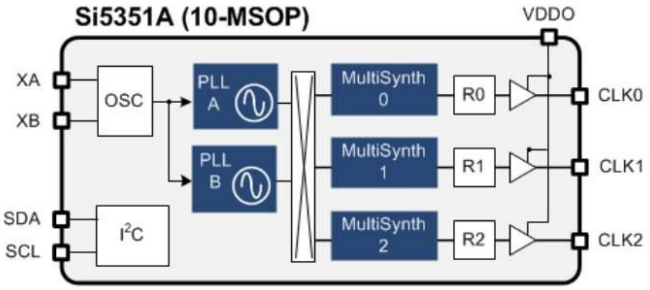


One (complex) input is multiplied with a “Wiggle factor” and either subtracted from or added to the other input. This is repeated over the complete array in $^2\log(N)$ subsequent stages, where the combinations and W factors change per stage. The result of each stage is stored in the same location, hence the notation “in-place”.

The FFT function has two ways to execute, in forward and in reverse, which are almost identical, except for a factor. In principle applying one after the other would result in the original samples.

3.3 User control processing

3.3.1 Quadrature VFO (si5351.c)



The Si5351A is a triple clock generator, that can be controlled through I^2C interface. There are three clock output stages, that can be driven by two PLLs. These PLLs multiply the crystal oscillator frequency (usually 25MHz) by some amount, from which the clock outputs are derived by another multiplication (division). Also, a phase offset can be given to a clock output, and when two clocks rely on the same PLL, the phase relation is deterministic. This characteristic is used to make a quadrature VFO, with two outputs with the same frequency but with controlled phase difference (0, 90, 180 or 270 degree). For the Q mixer input, a sin() is needed which has a -90˚ phase with regard to the I mixer input, which is a cos() signal. You can also say that the sin() is a quarter wave delayed cos().

The 90˚ phase difference can also be created in hardware. In the current µSDR-Pico implementation this is done by two fast flip-flops controlled with a VFO frequency of 2x the mixer frequency and with 180˚ phase difference. This double frequency has to be accounted for in the user interface.

The fractional multiplier for the PLL stage must be so, that the resulting frequency is between 600 and 900MHz (MSN is between 24 and 32). These boundaries are not very hard, and can logically be between 15 and 90, but for the moment let's stick to the prescribed range. The Multisynth fractional divider for clock i is MSi (8..2048), after which an additional division with an integer factor Ri (1..128).

The multiplier and divider are written as: a+b/c

The trick is now to use integer mode for MSi, meaning that this should be an (even) integer division. Only then the phase offset can be used to produce an exact phase difference.

So starting from mid-range PLL output (750MHz), you can set MSi and Ri to get into the ballpark desired output frequency. Then tuning can be done by changing the PLL multiplicator MSN:

- $F_{out} = F_{vco} / (MSi*Ri)$
- $F_{vco} = F_{xo} * MSN$

Some range extremes (vary MSi to get anything between):

Ri	MSi	Range [MHz]
1	4	150.000 – 225.000
1	126	4.762 – 7.143
32	4	4.688 – 7.031
32	126	0.149 – 0.223
128	4	1.172 – 1.758
128	126	0.037 – 0.056

In practise we use:

- Ri=128 for F_{out} <1 MHz
- Ri=32 for F_{out} 1-6 MHz
- Ri=1 for F_{out} >6 MHz

Two VFOs have been defined, VFO 0 (output on clk0 and clk1) and VFO 1 (output on clk2). A number of macro's have been defined to control the vfo:

- `SI_GETFREQ(i)` Returns frequency of VFO i
- `SI_INCFREQ(i, d)` Increment frequency of VFO i with d Hz
- `SI_DECFREQ(i, d)` Decrement frequency of VFO i with d Hz
- `SI_SETFREQ(i, f)` Set frequency of VFO i to f Hz
- `SI_SETPHASE(i, p)` Set phase delay of VFO i to (p = {0, 1, 2, 3} x 90deg)

Note: `SI_SETPHASE` obviously only works for VFO 0, where the delay is introduced in clk1.

The function `si_evaluate()` is called to evaluate whether VFO settings have actually changed and then write the new settings to the si5351 registers. That is more efficient than writing for every Hz when turning the tuning knob.

3.3.2 Display (lcd.c)

The display is a 16x2 LCD controlled with the familiar HD44780 chip, but the version used here is controlled over an I2C bus. This allows to also use for example an OLED graphical display instead.

The software driver contains a 16x2 byte buffer, which can be copied to the LCD in two write actions, when necessary. Of course, also characters can be written one after another. Also, the current cursor position is maintained with the buffer, this should normally match the cursor position on screen.

Apart from the initialization function, there are several other available to control the output:

- `lcd_ctrl()` Controls display state.
- `lcd_put()` Output one byte to current cursor position.
- `lcd_write()` Output string to current cursor position.

The output functions also move the cursor location horizontally, until the last column is reached.

The control function supports the following actions:

- `LCD_CLEAR` Clear display, cursor to left top position
- `LCD_HOME` Cursor to left top position
- `LCD_GOTO` Move cursor to x, y position
- `LCD_CURSOR` Set cursor visible or not
- `LCD_BLINK` Set cursor blinking or not

3.3.3 User interface (hmi.c)

The user interface is event driven, and organized around an IRQ callback routine. This handler catches the events on the GPIO pins used for the encoder, for the buttons and for the PTT. The interrupts are caused by rising and falling edges detected on the GPIO. From this the encoder increment and decrement events as well as the key-pressed events for the other buttons are deduced.

Events:

- Encoder increment
- Encoder decrement
- Enter key
- Escape key
- Left key
- Right key
- PTT activated
- PTT released

The HMI can be in several states, and depending on the state the events will have different effects. The top level is the normal operational state, which only allows tuning the operating frequency. From this top-level the sub-menu level can be entered by pressing the ESC button. There is only one sub-menu level.

In the sub menus a value can be changed with the Encoder and accepted with Return. Left and Right buttons cycle through the sub menus and pressing ESC again exits the sub-menu level.

Submenu States (to be expanded):

- Mode (USB, LSB, AM, CW)
- AGC (Fast, Slow, Off)
- Pre amp (+10dB, 0dB, -10dB, -20dB, -30dB)

Transmission

The PTT active event enables the TX path. The PTT event is directly associated with the HW PTT signal, that also directly controls HW like the BPF.

3.3.4 Command shell (monitor.c)

The command shell provides a command line interface on stdin/stdout. The Pico supports two mappings for stdio, either to the USB or to the physical UART0, selectable in CMakeLists.txt. In the final situation the idea is to provide a proper serial interface through the UART. This then also supports logging errors during the device start-up phase.

To enable stdio, a call has to be made to `stdio_init_all()`. This is actually done inside the monitor initialization routine, so best to call this early in the start-up sequence.

The shell vocabulary is contained in an array of strings. Whenever a CR/LF is entered on stdin, the collected characters are treated as command-line, and a match is attempted with the strings in the shell array. When a match is found, the corresponding handler is invoked, with the remainder of the command-line.

Handlers can be added where needed, for debugging or control purposes.

4 Hardware design

The µSDR hardware design is based on off the shelf modules and is installed in a Teco 1500 enclosure. It is an improvement of the previous version (3.0) but still serves as a test and experimentation environment. However, the modularity of the prototype is easily allowing adaptations, since you can swap out certain functions of the system that do not work appropriately. The main portion of the functionality is realized in software.

The modular hardware is structured as follows:

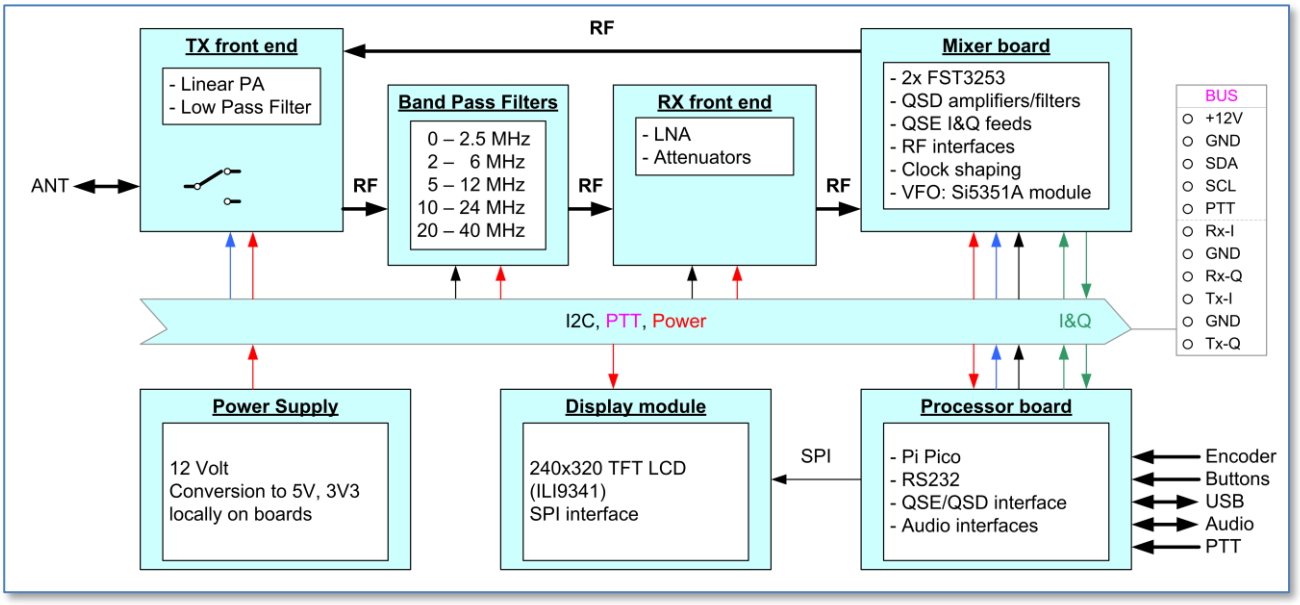


Figure 4-1 HW architecture, modules overview

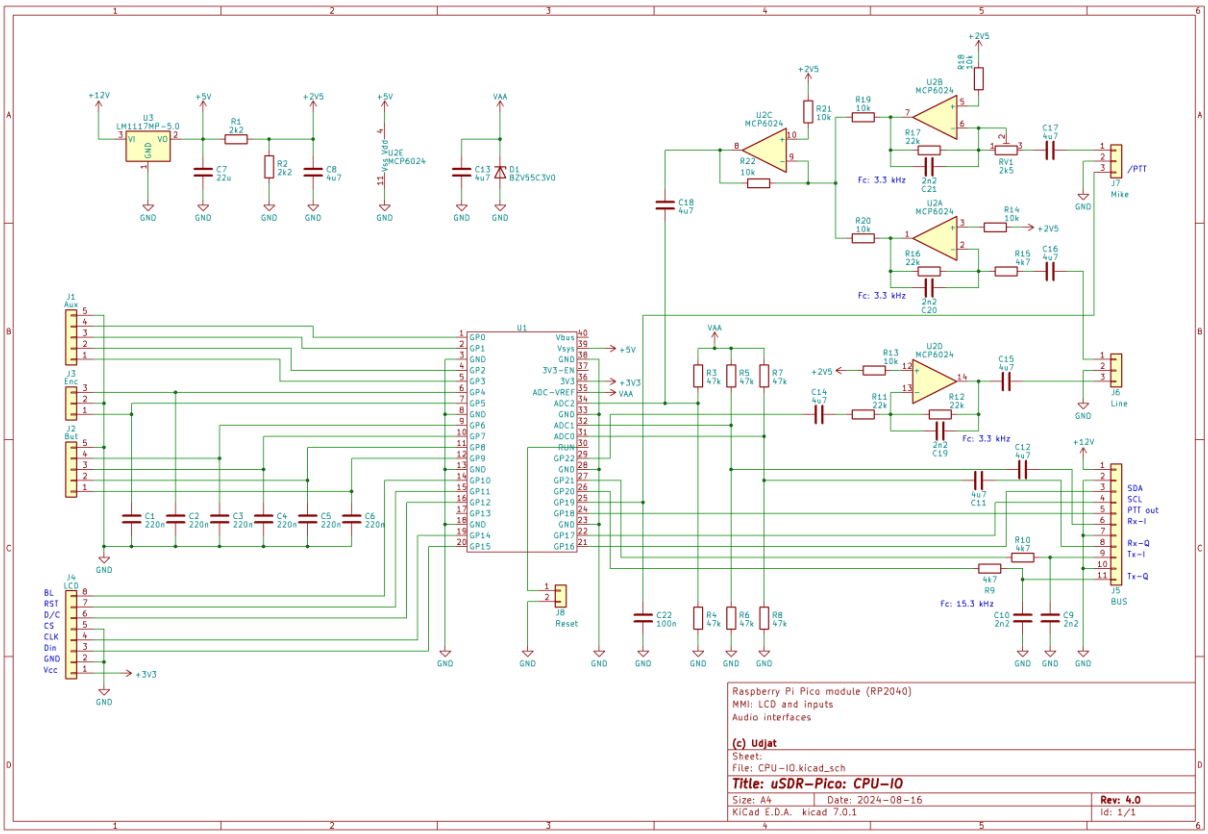
The signal path leads from Processor board to the Band Pass Filters and vice versa. Internal control is done through an I^2C bus and a discrete PTT signal. The latter is either passed through from a microphone, generated by the VOX or issued by SW in the Processor. The display is controlled through a SPI interface, so it will not interfere with the I^2C devices.

In summary, the modules are:

- Processor board: This is simply a carrier for the Pi Pico and the audio I/O circuits. The interfacing comprises the switch debouncing, PWM/DAC filtering and audio handling towards line or mike/headset. On the other side the quadrature inputs and outputs lead to the Mixer board.
- Display module: The display module can be anything suitable with a SPI interface. The SW is made for a 240x320 graphical GUI, based on an ILI9341 controller.
- Mixer board: This is a design based on two FST3253 multiplexers. The board also hosts the VFO, clock shaping is done with two 74AC74 dividers, and analogue IF signal shaping with LM4562 OpAmps.
- RX front end: This board contains the RX LNA and attenuators.
- TX front end: This boards contains the TX PA and a low-pass filter. Since the TX signal passes through the BPF board, the generated RF maximum power level is limited.
- BPF board: This contains a set of 5 switched bandpass filters, that go in between the RX front-end and the antenna switch. The switching between the RX and TX path is done by a PTT-controlled relay.

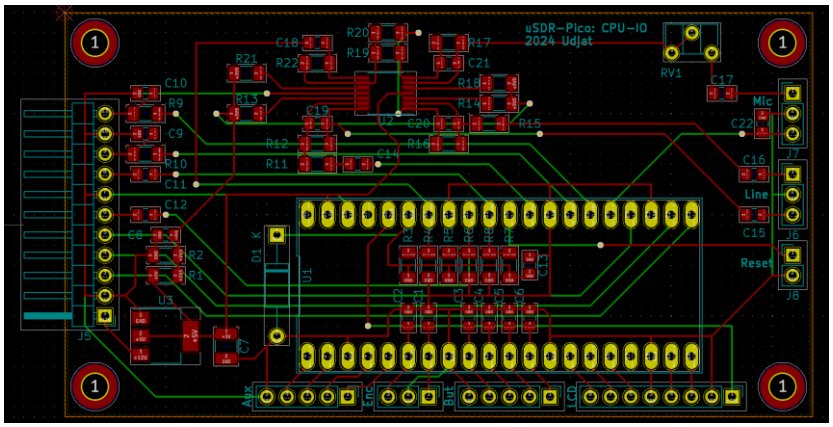
All boards have the same form factor, 2” x 3.7” (51x94mm), with co-located screw holes and can be mounted in a stack. In the following sections the version 4.0 is shown, which require some patches to function properly. An update 4.1 will be made available in KiCad that already contain those patches.

4.1 CPU-IO board

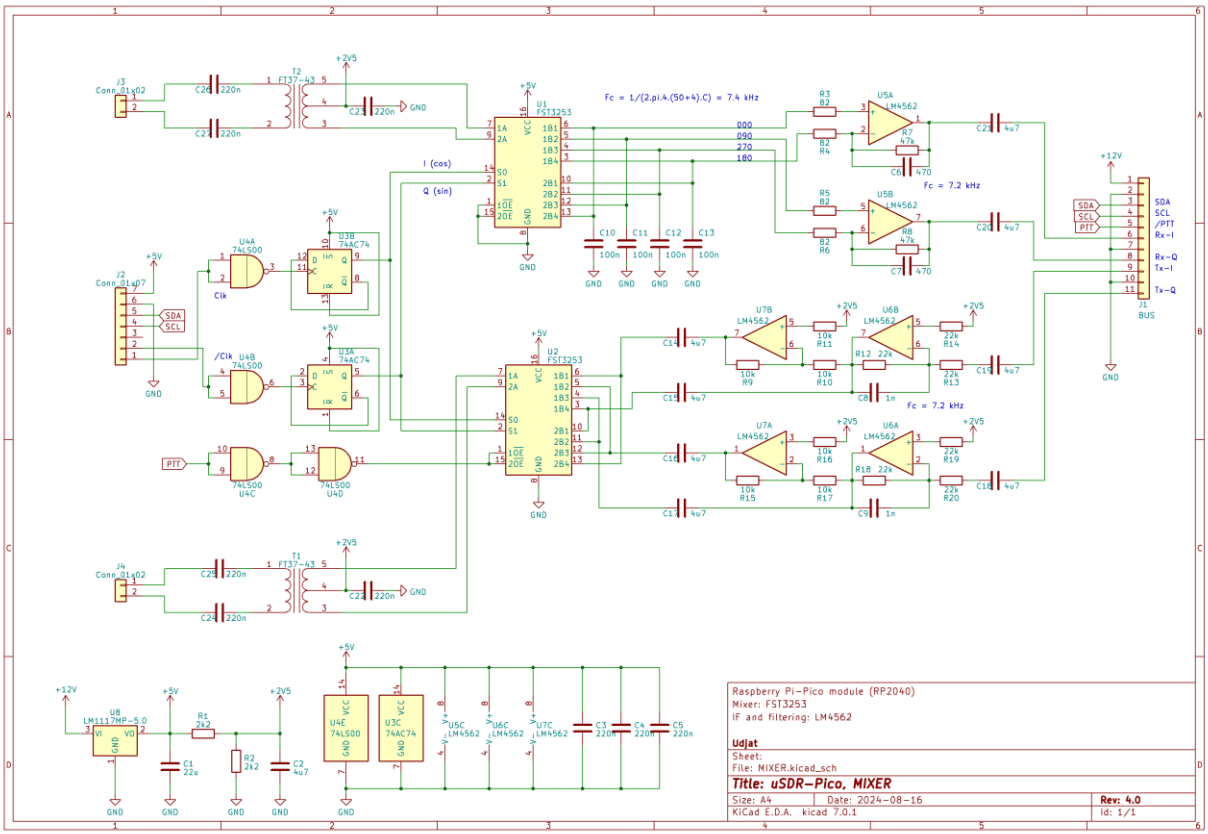


The CPU-IO board hosts the Pico module and the user interfaces, including the audio handling for mic input and line I/O. The USB interface on the module is made externally accessible to provide a monitor with a command line interface as well as the programming interface. The I and Q output low pass filters have a corner frequency of 15.3 kHz, all audio interfaces are low pass at 3.3 kHz. Four free GPIOs have been made available on a pinheader for potential later use. The bus connector interfaces some important signal to other boards: Power, PTT, I2C, I/Q signals for RX and TX.

Patches: 82Ω from C13 to 3V3 rail (to properly bias the 3V Zener).



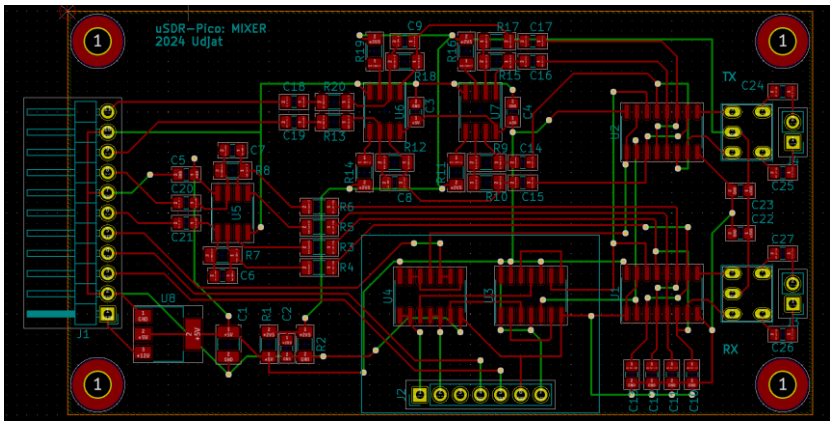
4.2 Mixer board



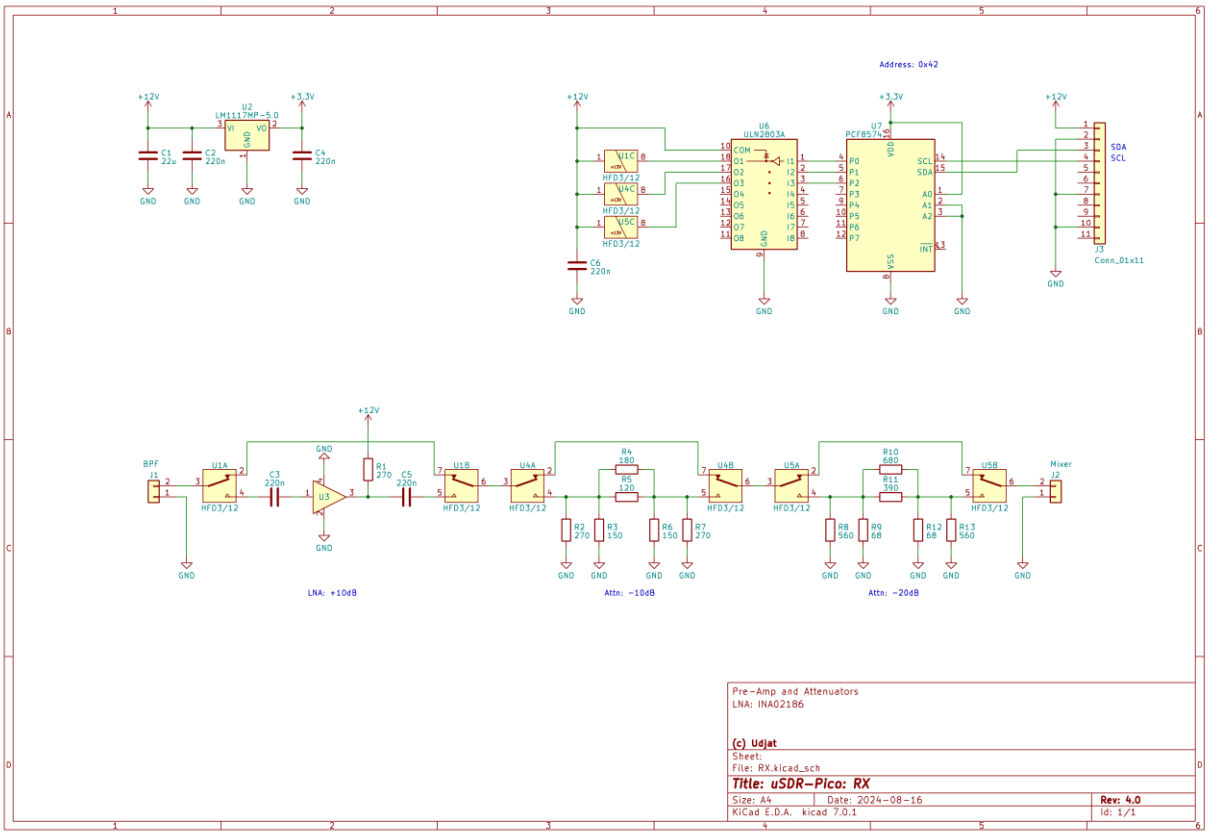
The Mixer board still hosts two FST3253 based mixers, QSD for RX branch and a QSE for the TX branch. The Si5351 VFO module is placed on the Mixer board. The low-pass corner frequency of both RX and TX path is set to about 7.2kHz, to allow for the center frequency shift required when doing FFT based signal processing. The TX path is disabled when PTT is not active (i.e. logic '0'), the RX path is always enabled.

Patches: Cut output of U4C at pin 8 and connect pin 9 with 12/13 to reverse PTT active (1 instead of 0). Remove U3 dual flipflop and connect U4 pin 3 to U3 pad 5, and U4 pin 6 to U3 pad 9, to restore quadrature clock possibility from Si5351. The original design with 180˚ difference and flipflop dividers, intended to improve I-Q phase relation, actually gave undefined results (I-Q swapped at times).

Note: When the sampling (timeslot) frequency in the Pico is increased, the bandwidth expected from the Mixer is also increased. In that case the low pass filters must be adapted accordingly.

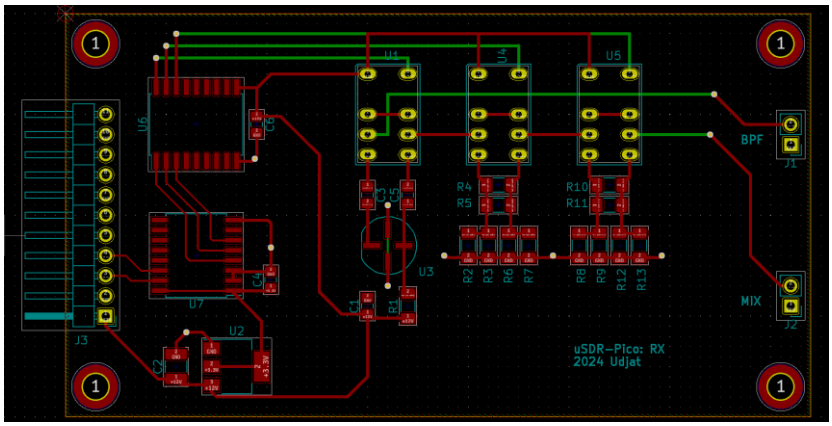


4.3 RX board

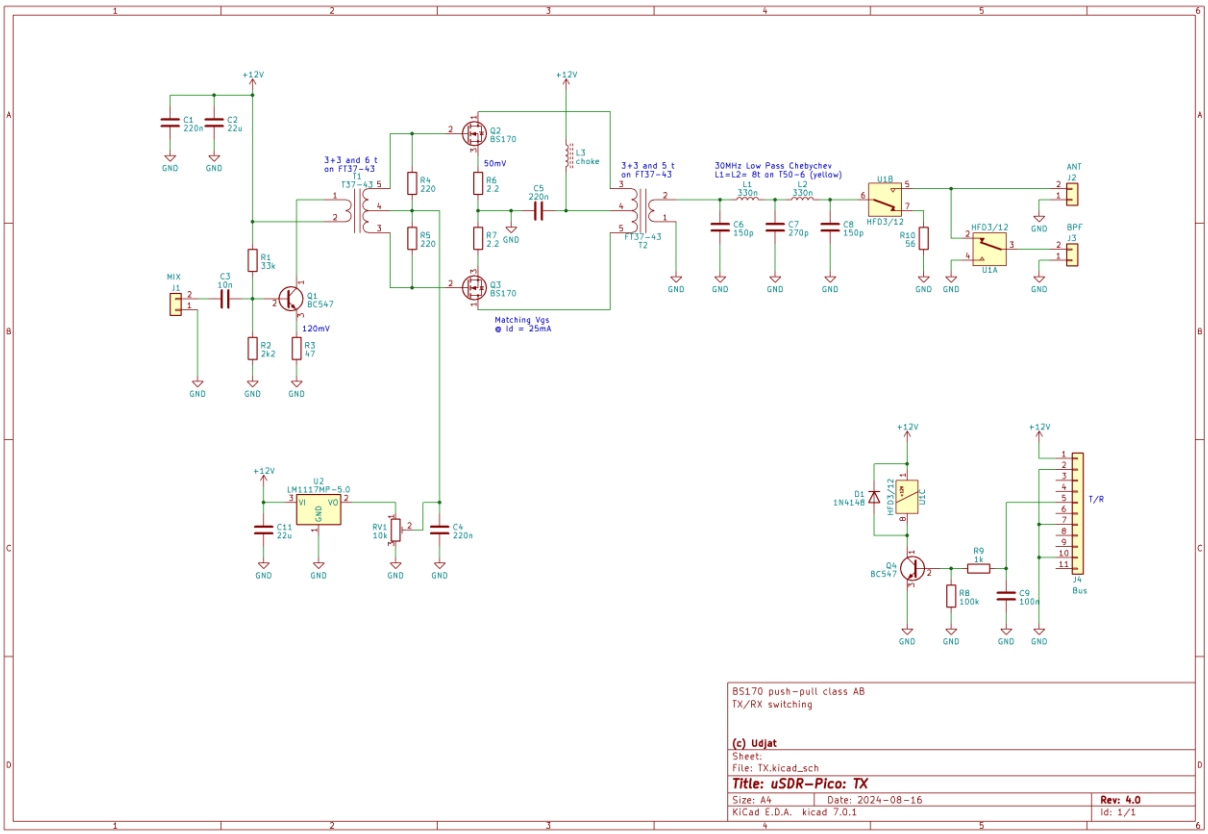


The RX board just contains the selectable attenuators and a low noise amplifier, to obtain settings in 10dB steps. The INA02186 will deliver about 10dB gain when the supply current is approximately 24mA.

Patches: Add 1k2 pullups on the five signals between U7 and U6. Drive capacity of the PCF8574 is a bit too low for the ULN2803.



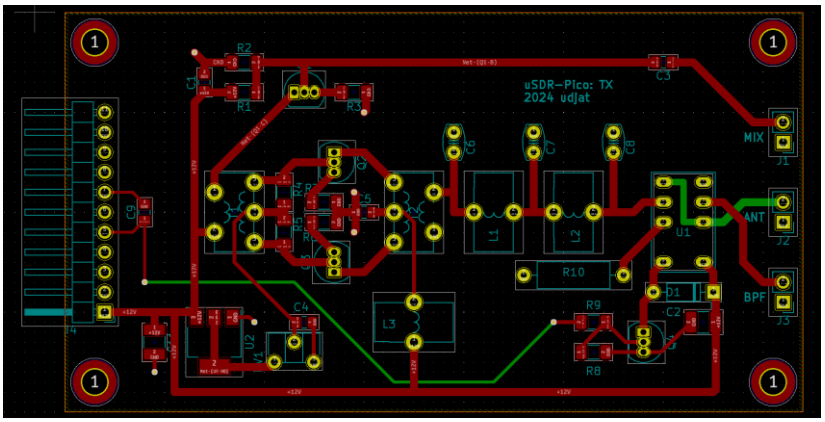
4.4 TX board



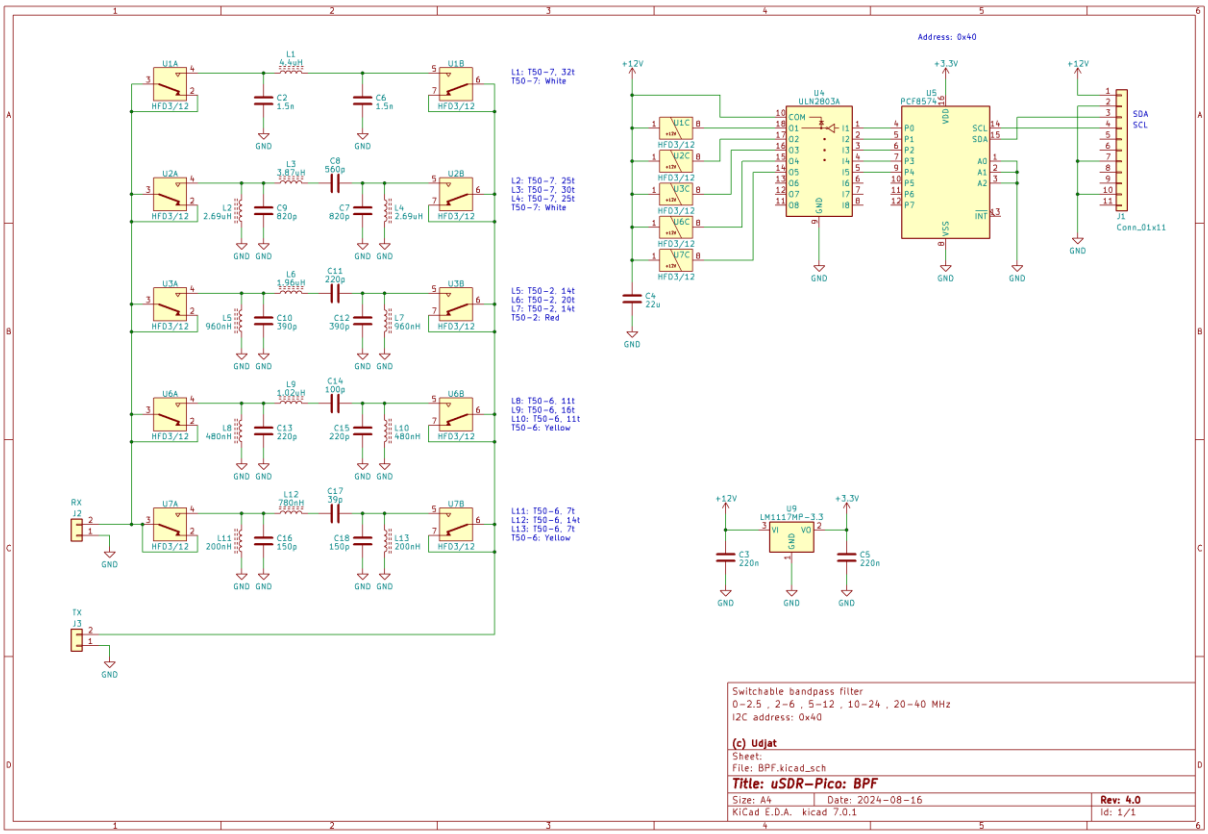
The TX board is a very simple class AB, BS170 based push-pull amplifier, driven by a regular class A stage. The PA should be able to deliver about 500mW.

Bias is adjusted to just above cutoff, so a standing current of about 20-25mA per MOSFET drain, to be adjusted for signal symmetry when the amplifier is operational. It is recommended to match the BS170 pair for gate threshold voltage. Matching can be done by connecting gate with drain and feeding the drain from 12V through a 330Ω resistor. V_{gs} must be the same for a match (around 3V).

This is just an example PA, which needs to be further tested and could be replaced easily by another should this be necessary. Bear in mind to keep the output power within limits, since it has to be taken through the BPF banks.

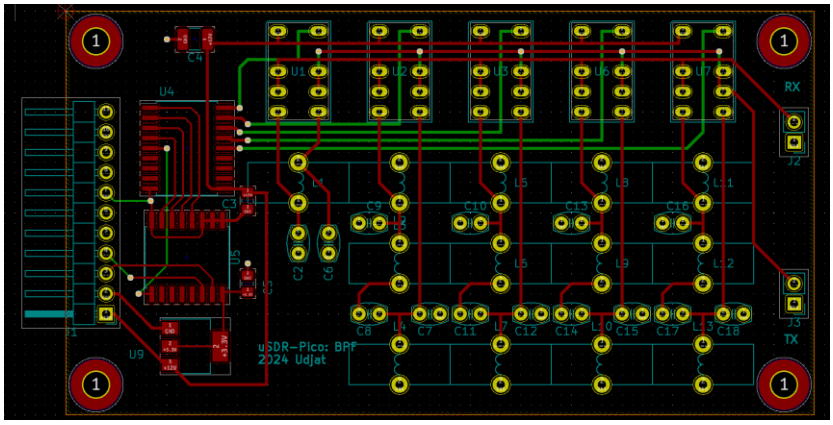


4.5 BPF board

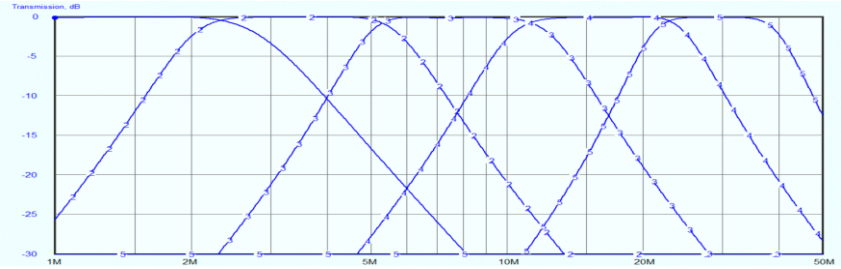


The 5 band filters are selected through relays, controlled from the Pico via the I^{2}C bus. A sixth relay is connected to the PTT signal, and connects either RX or TX path to the filters.

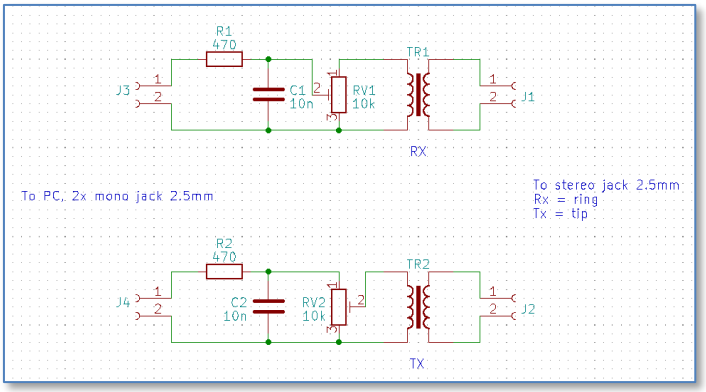
Patches: Add 1k2 pullups on the five signals between U5 and U4. Drive capacity of the PCF8574 is a bit too low for the ULN2803.



The (roughly octave) filters are calculated with ELSIE, and have the following pass-through characteristics:



4.6 Other



This circuit is used in the cable that connects PC with uSDR-Pico, and serves as galvanic separation; PCs can be rather noisy… The trimmers can be used to adjust the required levels.

4.7 Mechanics

The mechanical construction matches the pin-headers/connectors on the different PCBs for an optimal wiring. The only error at present is the power and I2C connections on the BPF board, which should ideally be on opposite sides of the board.

The lot will be built into a Teko Euro93 series enclosure, type 936. The bottom has a partial aluminum base-plate, onto which the PCBs are mounted. The available space will be organized roughly as follows:

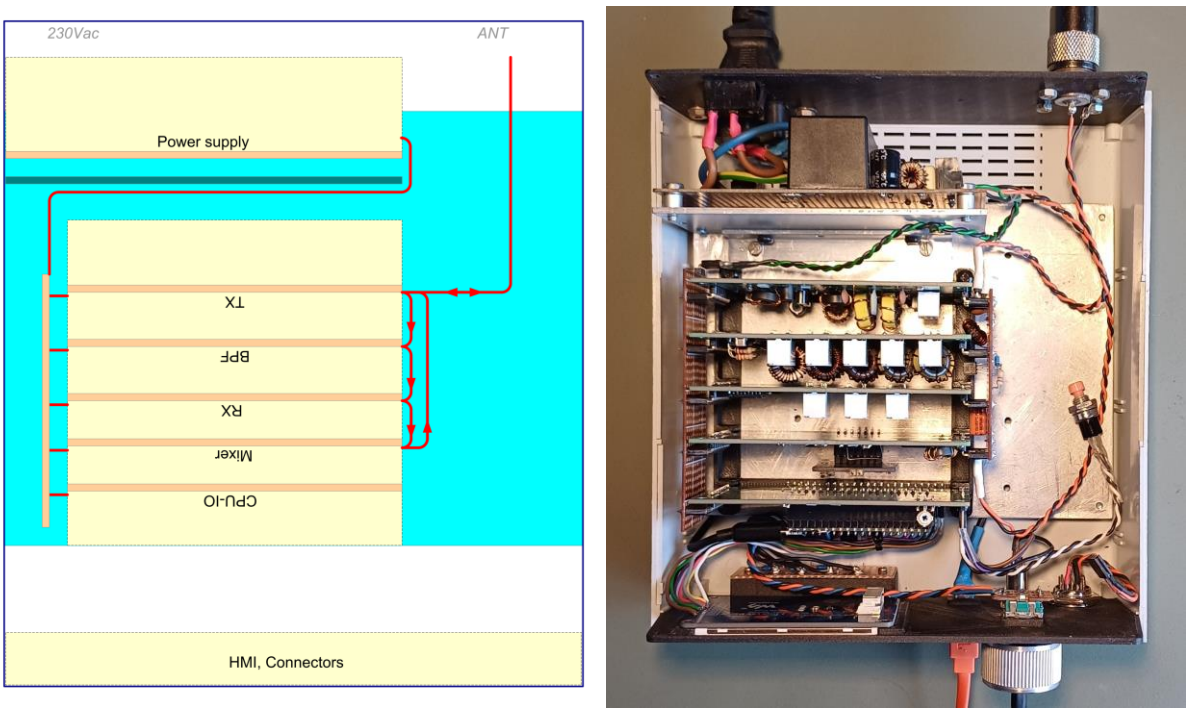


Figure 4-2 µSDR-Pico Next mechanical layout

On the left side, a bus board interconnects the SDR boards. This also connects to the 12V PSU in the back.

On the right side, another board is used to conenct the analogue IF and RF signals. This also contains a feed for the active loop antenna I use.

I have 3D printed a holder with slots for the PCB stack.

As can be seen, it leaves plenty room for some additions, like a more powerful PA stage with VOX switch.

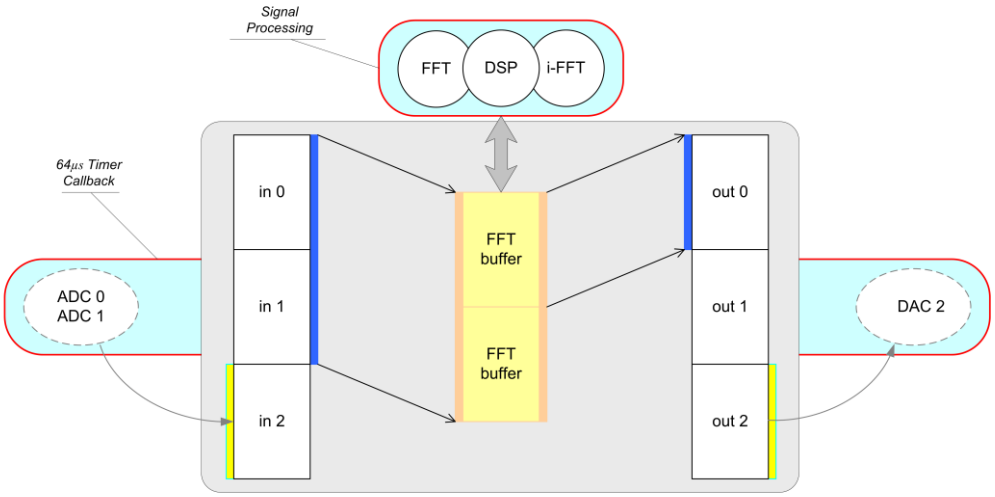
Annex A: Frequency domain processing

Overview

Another approach is to use frequency domain signal processing. This can be accomplished by transforming the stream of time-based samples into a spectrum. Sticking to the sample rate used in the ***uSDR*** v2.0 implementation, 15625 integer based *complex (I,Q) numbers* per second, this could be chopped up into suitably sized chunks for FFT. The chunk size is determined by the spectrum resolution that needs to be achieved. When for example a 1024 length FFT is used, the frequency resolution will be the sample rate divided by the FFT length (i.e. 15Hz). The 1024 point spectrum resulting from the FFT holds frequencies from DC up to half the sample rate (Nyquist).

Most signal processing can then be done in the frequency domain, where for example a bandpass filter can be implemented as a window over a portion of the spectrum. The time domain can finally be restored by applying an inverse-FFT, resulting in a processed (complex) sample stream at the original input rate of 15625.

To address aliasing issues the transformations are done with some overlap, for example every 512 samples for a 1024 FFT length. This means that every 512/15625=32 msec the 1024-point FFT, the signal processing and the iFFT need to be executed plus some overhead to manage the data and create the intended output.



In order to optimize for speed, the implementation will use fixed-point arithmetic, since the Pico features no FPU. For buffering the data, three 512-sample input buffers are required, where one buffer is being filled by the ADC while the other two are used as FFT input. Likewise for the output, where two buffers are target of the i-FFT, while the third is used to hold the DAC output samples. Intermediately, a 2x512 sample buffer is required to hold the spectrum. The buffer assignments are changed every 512 samples. Knowing that each sample contains a 2x16-bit complex value, this brings the total RAM use to 8x512x2x2=16kBytes, which should easily fit into the available 256kB.

Physical meaning of the FFT

An RF signal is mixed down with a quadrature mixer, resulting in an in-phase (I) and a quadrature (Q) signal, centered on DC. So what does this mean, for example to have negative frequencies? If you consider the original RF signal having two sidebands from amplitude modulation, the sidebands are just a bit higher or lower than the carrier frequency. When you mix this signal down with the carrier frequency, the lower sideband as a mathematical consequence is represented by a negative frequency.

Suppose that the mixed down signal is represented by the time dependent vector (I_t,Q_t). The rotation speed of the vector represents the frequency, the rotation direction represents whether the frequency is positive or negative. The actual movement of the vector is erratical, and contains a whole set of frequencies. With the fourier transform we actually want to analyze for this set of speeds (frequencies) to what extent these are present in the (I_t,Q_t) stream.

To this purpose, the I and Q streams are converted in discrete (I_k,Q_k) sample pairs, which are the time-domain complex input to the FFT. At regular moments in time the latest N samples (i.e. the FFT size) are transformed into a frequency spectrum. This transformation interval has to be shorter than what is represented by N samples, so there is some overlap.

2N *real* time samples would result in N *complex* frequencies (cf. Nyquist), where the missing negative complex frequencies are just a mirror of the N positive frequency bins. In contrast, 2N *complex* time samples result in N positive + N negative *complex* frequencies.

The bin with index 0 represents the DC component, and the bin with index N-1 represents the Nyquist frequency. The bins N and beyond represent the negative frequencies.

For representation, rotating the set of frequency bins with N will place the DC component at bin N and the upper/lower sidebands on either side.

Demodulation of SSB would boil down to filtering away everything but the 3kHz or so to the right of the DC bin before transforming back to the time domain. Since Upper and lower sidebands are different, it is clear that complex time samples are required to obtain the right transformation.

To get rid of noise around DC, the mixing frequency could be chosen lower than the actual RF carrier, causing the baseband signal to land somewhere in the middle of the positive frequency bins, i.e. around N/2.Then after filtering, the baseband can be shifted down by N/2 before applying the inverse FFT.

The FFT mathematics

The Fast Fourier Transform is an optimized implementation of a Discrete Fourier Transform. This transform changes a series of time-domain samples into a frequency spectrum.

The DFT equation is given by:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot e^{-j\frac{2\pi}{N}nk}$$

For the inverse operation (i-DFT) a similar equation applies:

$$y(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) \cdot e^{j\frac{2\pi}{N}nk}$$

The DFT represents a multiplication of a *size-N* time samples vector with a *size-N^2* square matrix, resulting in another *size-N* frequency spectrum vector. The samples are in fact the I and Q streams coming from the QSD, and can be represented as:

$$x_n = I_n + j \cdot Q_n$$

The complex coefficients of the DFT can be represented as:

$$e^{-j\frac{2\pi}{N}nk} = cos\left(\frac{2\pi}{N}nk\right) - j \cdot sin\left(\frac{2\pi}{N}nk\right)$$

In these complex coefficients **n** represents the time, indicating one element of the sample vector, while **k** represents the frequency, indicating one element of the spectrum vector. Increasing **k** means increasing frequency, i.e. the number of full phase waves that fit inside the total sample period. The factor **n/N** steps timewise through this period. Hence the internal product between a matrix row and the sample vector is calculated for each possible frequency.

Each *complex* multiplication in that internal product in fact consists of 4 multiply actions and 2 additions:

$$(I_n + j \cdot Q_n) \cdot \left(cos\left(\frac{2\pi}{N}nk\right) - j \cdot sin\left(\frac{2\pi}{N}nk\right)\right)$$
$$= \left(I_n \cdot cos\left(\frac{2\pi}{N}nk\right) + Q_n \cdot sin\left(\frac{2\pi}{N}nk\right)\right) + j \cdot \left(Q_n \cdot cos\left(\frac{2\pi}{N}nk\right) - I_n \cdot sin\left(\frac{2\pi}{N}nk\right)\right)$$

So, to obtain one element of the spectrum vector **X(k)** this complex multiplication has to be performed **N** times as well as **2N** more additions.

In case of ***uSDR*** both **n** and **k ϵ [0, N-1]** and hence this results in N^2 *complex* multiplications and additions. However, the FFT implementation of the DFT utilizes the inherent symmetries in the operation to avoid doing duplicate multiplications.

These symmetries originate from the periodicity of the **sin()** and **cos()** functions in the DFT coefficients, where half a period of phase shift just results in a sign flip. Repeatedly applying this optimization for all **N** elements of the samples vector reduces the number of multiplications per element of the resulting spectrum vector from **N** down to $^2\log(N)$ and by doing so reduces the total number to $N \cdot {}^2\log(N)$. To illustrate this, for a 1024 size array the CPU burden for calculating a FFT would be approximately 100 times less than for a DFT.

A recursive FFT implementation could be applied by repeatedly splitting the input sample array in halves with a fixed phase difference. A more realistic implementation for ***uSDR*** is not going to be recursive however, in order to prevent spending too many cycles on moving data around. Instead, the buffer mechanism as described in the overview is used, such implementations are referred to as in-place. The input array is copied to the output location and then the storage of that copy is used to perform the actual transformation, in-place.

Several examples of fixed-point FFT implementations in C can be found, but any example needs to be optimized and adapted to the specific target environment. For ***uSDR*** the first go is based on **`fix_fft.c`**, originally written in 1989 by Tom Roberts but since then improved several times by others. Another good source is *fxtbook.pdf*, to be found on the internet. This latter algorithm is used in ***uSDR***.

FFT implementation

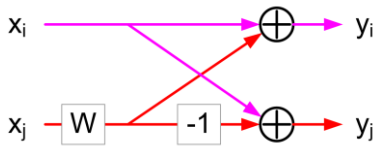
The first stage of the FFT algorithm is the reordering of the samples; to be precise, the samples with indexes that are bit-reverse of each other need to be swapped. The array size is 1024 samples, so the index is 10 bits. The swap is then for example between samples [1111000001_b] and [1000001111_b]. This re-ordering is done before the FFT is calculated, and therefore named Decimation in Time (DIT), as opposed to the post-FFT DIF. The re-ordering is needed to enable an efficient chain of butterfly executions.

The crux of the algorithm is how to go through the array only once, i.e. avoid swapping samples back again. A general approach would be:

```
for (i=0; i<1024; i++)
{
    j = bit_reverse(i);
    if (i < j)
        swap(data[i], data[j]);
}
```

but the `bit_reverse` routine could take quite some time. A faster alternative (utilizing the relatively abundant RAM) is to use a lookup table instead. After that, the swapping of the samples is straightforward.

The second stage consists of a number of nested loops, calculating and adding the butterflies. One such butterfly can be represented as follows:



One (complex) input is multiplied with a "Wiggle factor" and either subtracted from or added to the other input. This is repeated over the complete array in $^2\log(N)$ subsequent stages, where the combinations and W factors change per stage. The result of each stage is stored in the same location, hence the notation "in-place".

Sequencing

ADC read and DAC write

The ADCs are running in RR mode, just like in the time domain uSDR implementation. The difference is that after 3 samples the conversions are temporarily stopped. It means that for 3x 2usec cycles conversions are done for ADC channels 0..2 , and the result is shifted into the FIFO, flagging an IRQ at level 3. The ADC-FIFO interrupt handler will stop the conversions, which will be restarted by the timer callback routine. This way the sample frequency is 16.625 kHz.

The samples are copied from the ADC FIFO and stored in the related sample buffer by this timer callback. The callback routine also handles the output to the DACs from the related sample buffer. When a half FFT_SIZE buffer is full, the DSP-loop is signaled, which performs the FFT, DSP and iFFT operations during a ½ FFT-size x 64usec interval, which is 32.768msec for a 1024 FFT-size.

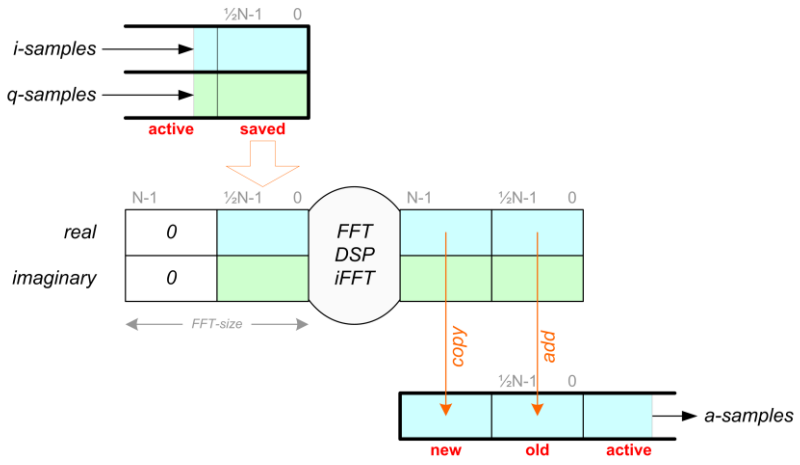
The signal should be amplified to fill the FFT dynamic range, i.e. up to about the range of `int16_t`. To do this, an average signal strength indicator is calculated so that a multiplier (call it an AGC) can be deduced. Normally the input signal is less than $1V_{pp}$, which is 1/3 of the ADC range of 12 bits, so the multiplier will be approximately 100.

A similar process must be applied to the output, because the DAC range is only 8 bits. Realistically only a part of the signal remains after processing, so again a sgnal strength indicator must be maintained to calculate a multiplier.

For a relatively smooth signal the peak signal strength indicator can be approximated with 2x the average absolute value.

Buffer handling

The buffer structure is built up from ½ FFT-size buffers, doubled for the complex side of processing. The Overlap-Add method is used to ensure a smooth glueing of the chopped-up sample streams.



The figure represents the RX case, the allocated buffers are in fact re-used but work in opposite direction for the TX case. The active interface buffer is one of a 2-buffer queue, the other being the saved samples of previous interval. The active buffers collect the I and Q samples captured by the timer callback routine. Whenever the active buffer is full, the input and output buffers are reorganized and the DSP loop is signaled to start.

The real part of the previous signal processing cycle result is added/copied to the output buffers. Then the saved input buffers are copied into the lower half of the FFT buffers, while the upper halves are zero padded. Then the signal processing cycle is begun, yielding a new result.

Overlap-Add method

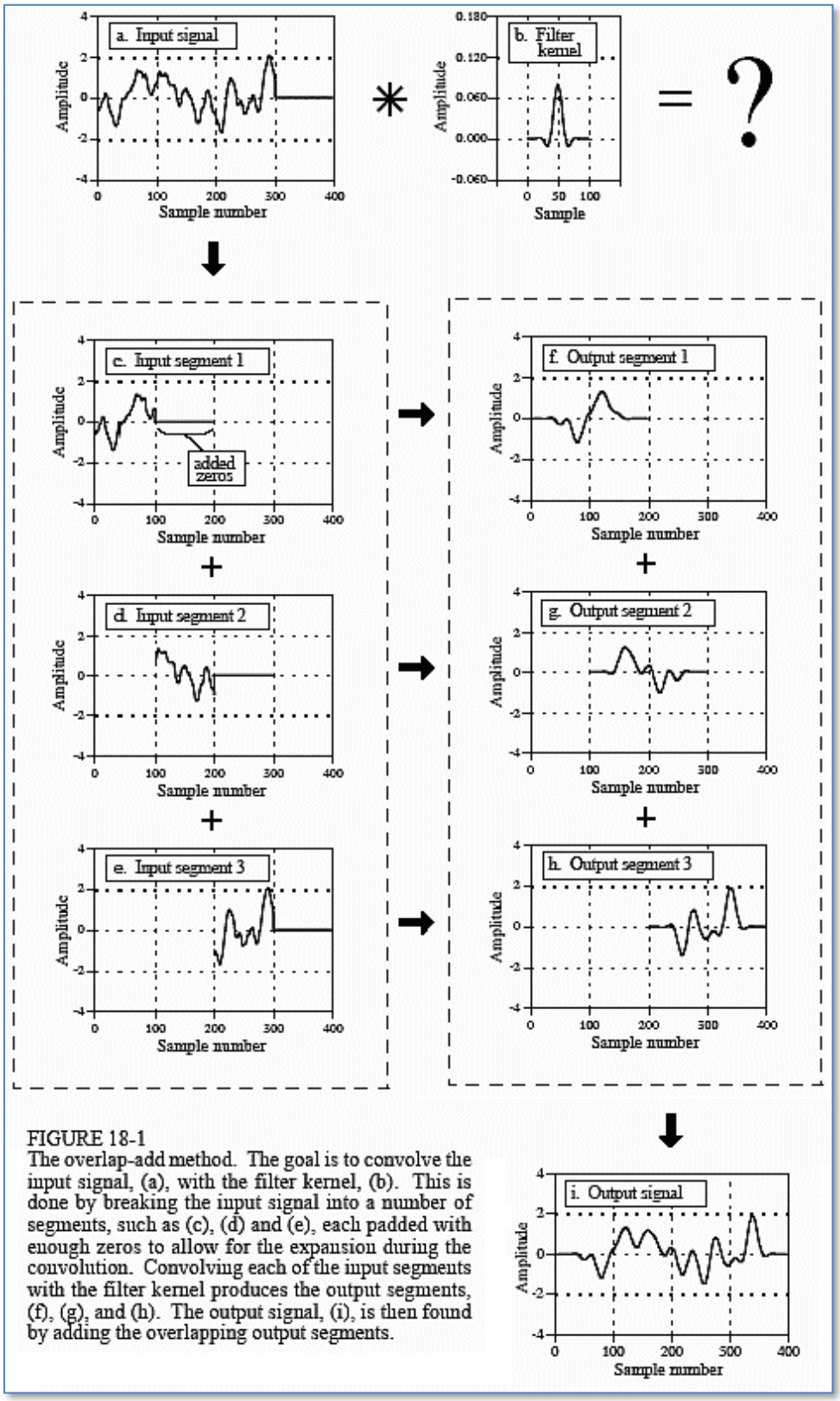


FIGURE 18-1
The overlap-add method. The goal is to convolve the input signal, (a), with the filter kernel, (b). This is done by breaking the input signal into a number of segments, such as (c), (d) and (e), each padded with enough zeros to allow for the expansion during the convolution. Convolving each of the input segments with the filter kernel produces the output segments, (f), (g), and (h). The output signal, (i), is then found by adding the overlapping output segments.

Source: https://www.eetimes.com/fft-convolution-and-the-overlap-add-method/