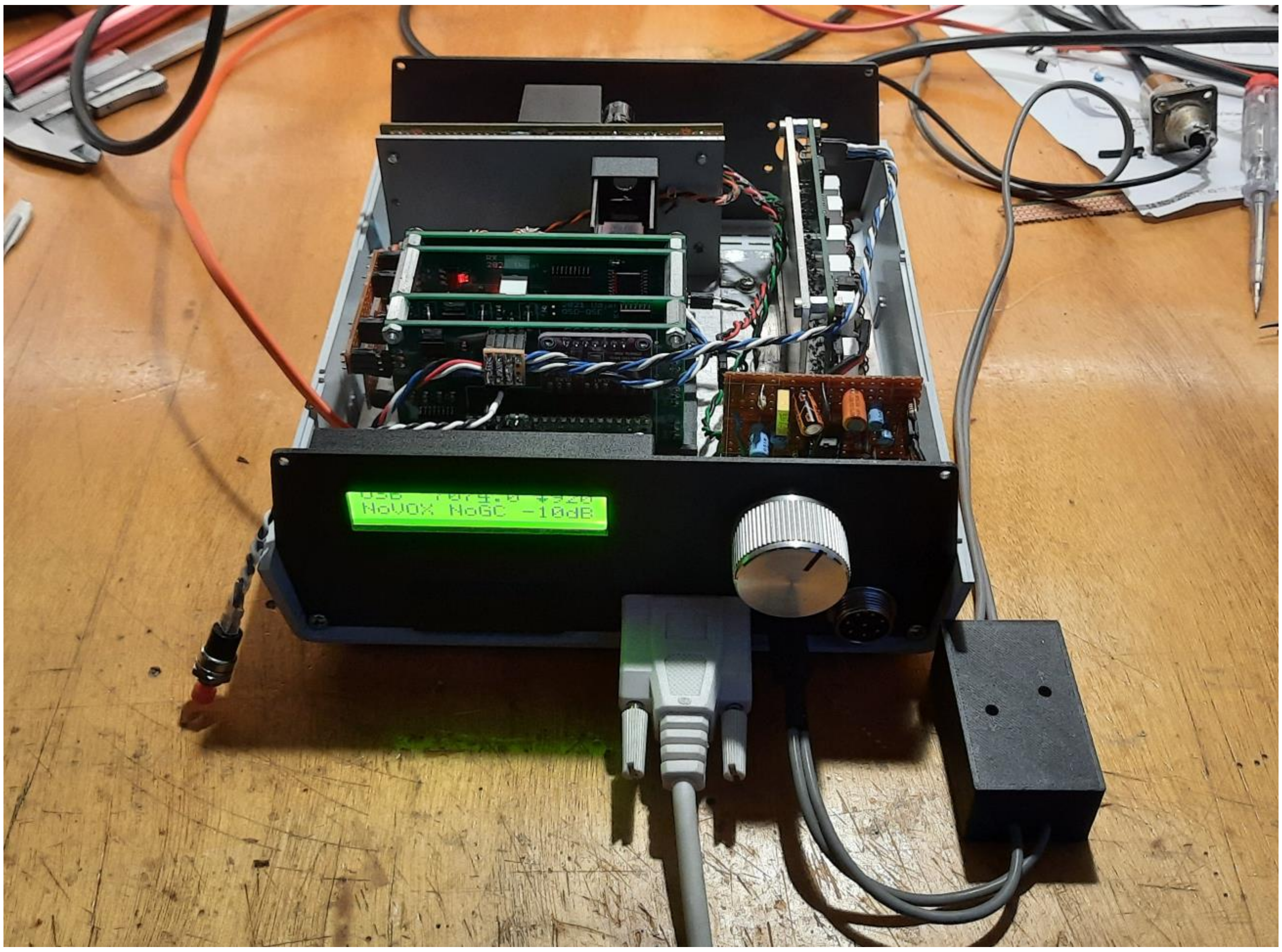
Micro SDR on a Pi-Pico

Arjan te Marvelde, Version 3.00 – June 2022 (initial version May 2021)



Since 2020, Raspberry offers a new module based on their own developed processor, the RP2040. This processor contains a dual core, 125MHz Cortex based controller with plenty of Flash and RAM. It has many highly configurable I/O pins, which make application of this module a breeze. The C-SDK does not seem to be fully mature yet (in May 2021) but at least it provides a quick start in actually setting this Pi-Pico to use.

This document describes a test implementation of a small SDR, inspired by Hans Summers' QCX and everything that followed after that, such as µSDX. A main deviation in hardware is the use of 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 scematic.

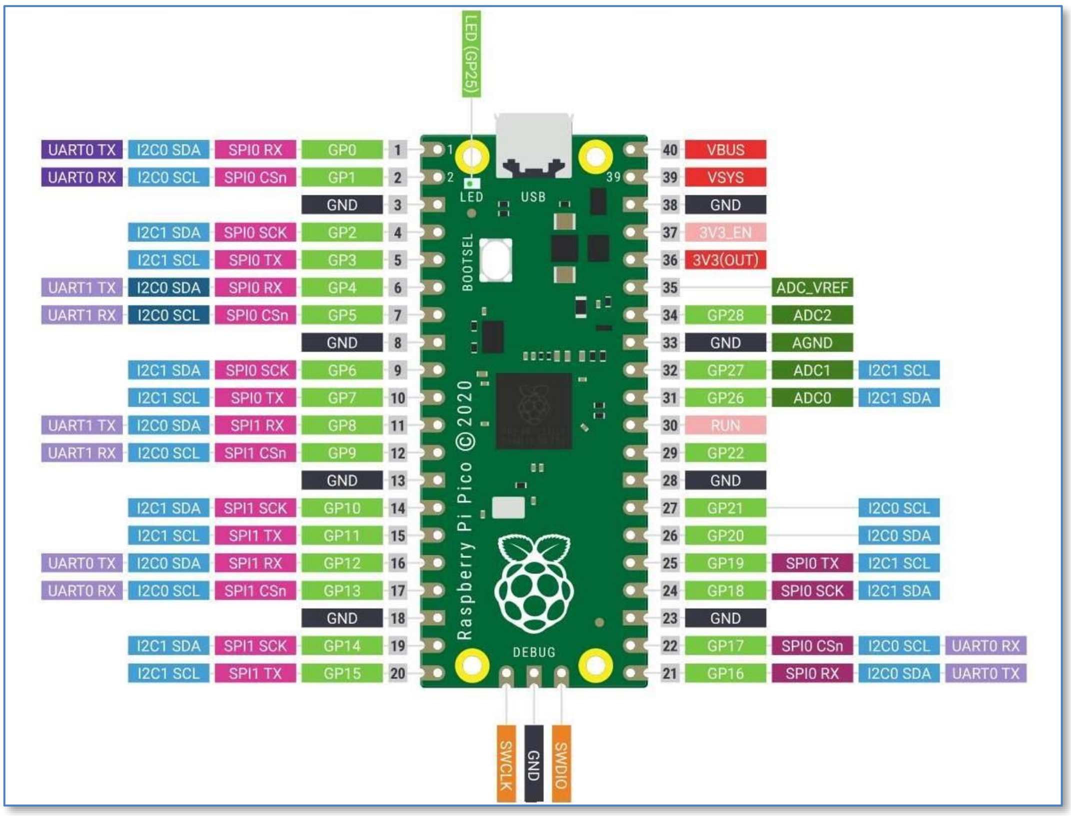
The image above shows the test setup at date of writing, where the RX an TX parts are working and modules are built into a Teko enclosure. Since v2.02 several bugs have been resolved, and the code has been restructured to allow insertion of the new FFT-based frequency domain signal processing.

Note to builders: This project is highly experimental and will remain a work in progress, I try to spend some time on it every now and then but it is just 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 maintained on GitHUB: https://github.com/ArjanteMarvelde/uSDR-pico. This version has been tested, and is ready to be used as starting point for your SDR experiments.

1 Raspberry Pi Pico (RP2040)

The Pi-Pico module is the heart of the implementation, so here is the pinout and the way it is used in this project.

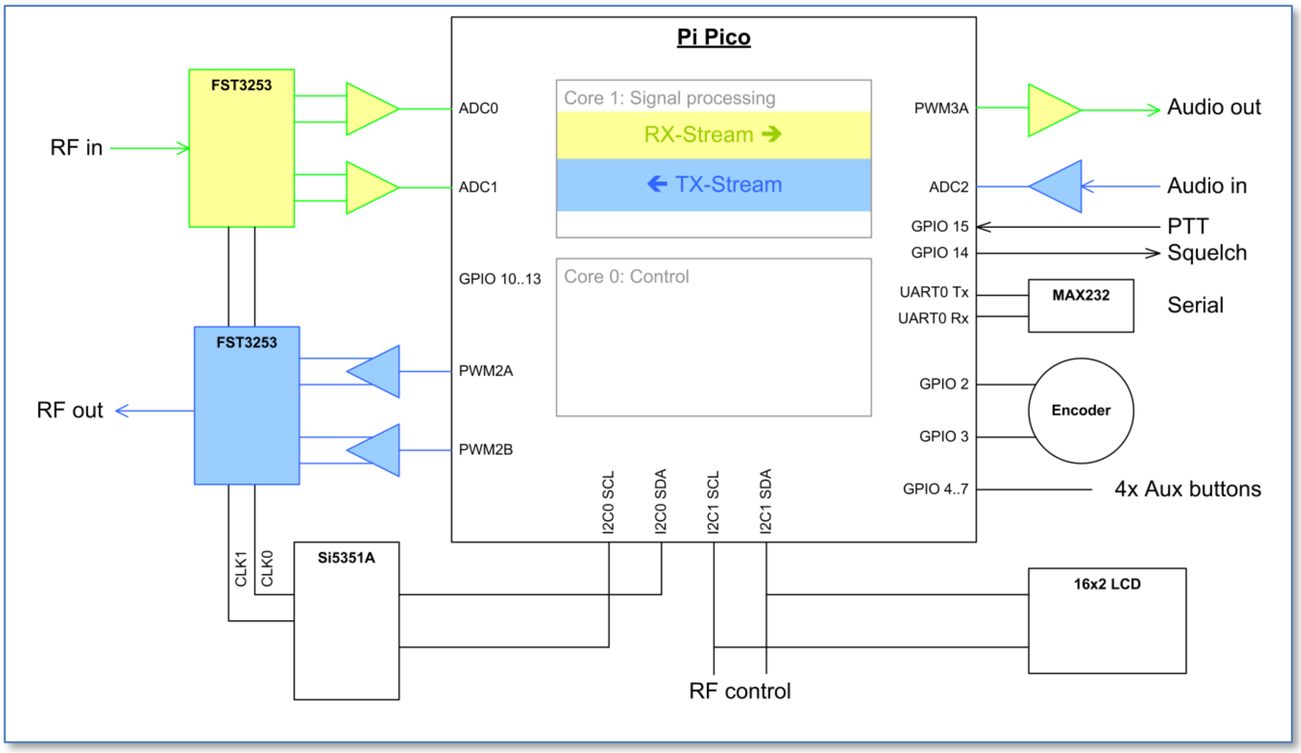


Pi Pico pin usage for uSDR project

UART0 Tx	🟦	GP0	Vbus	🟥	Not used
UART0 Rx	🟦	GP1	Vsys	🟥	5V power input
	⬛	GND	GND	⬛	
Encoder A input	🟨	GP2	3V3 en	⬜	Not used
Encoder B input	🟨	GP3	3V3 out	🟥	3V3 to peripherals
Not used	🟨	GP4	Va ref	🟥	ADC 3V3 reference output
Not used	🟨	GP5	GP28	🟧	ADC2: Audio input
	⬛	GND	GND	⬛	
Aux button 1 input	🟨	GP6	GP27	🟧	ADC1: QSD I-channel input
Aux button 2 input	🟨	GP7	GP26	🟧	ADC0: QSD Q-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	⬛	
	🟨	GP10	GP21	🟧	PWM 2B: QSE I-channel output
	🟨	GP11	GP20	🟧	PWM 2A: QSE: Q-channel output
	🟨	GP12	GP19	🟩	I2C1 SCL: LCD screen, BPF
	🟨	GP13	GP18	🟩	I2C1 SDA: LCD screen, BPF
	⬛	GND	GND	⬛	
Squelch output	🟨	GP14	GP17	🟩	I2C0 SCL: Si5351A
PTT input	🟨	GP15	GP16	🟩	I2C0 SDA: Si5351A

2 Principle of operation

The block diagram shows the processor board and several peripherals. The VFO is based on a **Si5351**, which may be for example 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 anything can be used that produces a quadrature RX signal and consumes a quadrature TX signal. There are 4 GPIOs reserved for band filter switching.



Note that the ADCs (and DACsO have a maximum range of 3V3 and should be limited accordingly.

On the user side, there are interfaces for Audio, PTT/Squelch, a Rotary Encoder, 4 (up to 8) Auxiliary Buttons, a serial port for a PC interface and another I^{2}C interface to control LCD as well as the RF front end (BPF, LNA, Attenuation).

Of course there is the USB interface which is used for device programming, and this could also be used as stdio based debug monitor instead of the UART on pins 1 and 2.

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 signal processing, while *core0* (the default at startup time) does all the 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). 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^{2}C buses, so we can make it easy and connect a fast one dedicated to the Si5351 and use the other to control the remaining devices. 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.

Files overview:

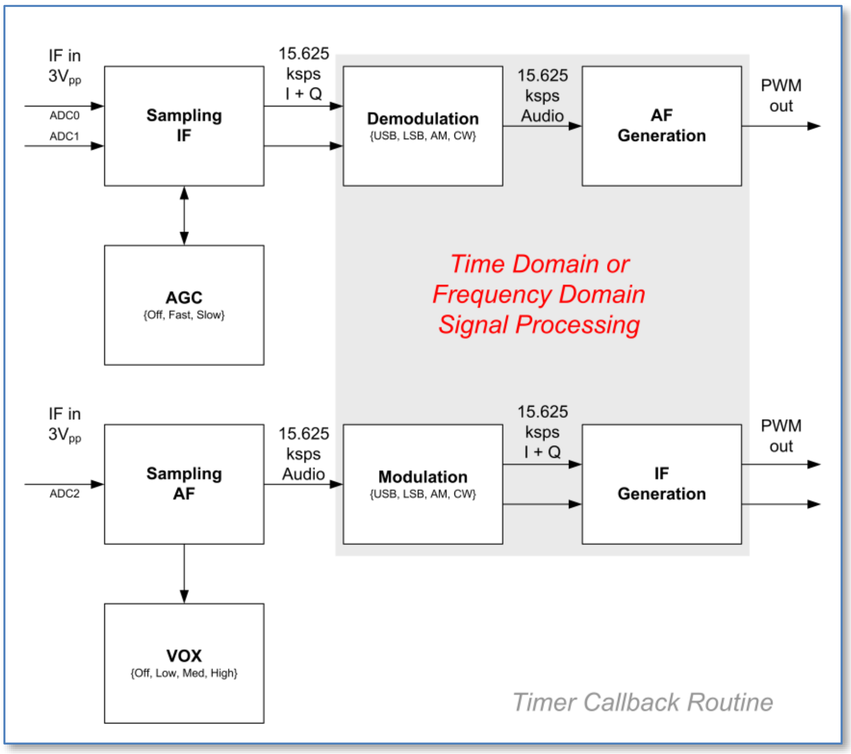
uSDR.c	The main loop and system initialization. The main loop runs as a separate process on *core0* and takes care of all user interfacing.
dsp.c	Signal processing loop, handles the sampling and invokes TX and RX branches. This loop runs as a separate process on *core1*.
dsp_tim.c	Time domain signal 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^{2}C expanders.

3.1 Signal processing

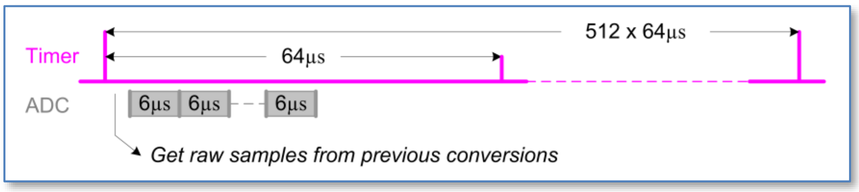
Where all other parts are enabling or control functions, the signal processing forms the heart of the uSDR-Pico. This is what this project is really about.

3.1.1 *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 constant in *dsp.h*. The main part in *dsp.c* contains the functions to acquire the samples, to do some preprocessing like level detection for AGC and VOX and further take care of the timing. This timing obviously works different in both methods; in the time-domain (TD) case the RX stream needs to be invoked for each sample and in the frequency domain (FD) case this is only every time an FFT buffer is filled.



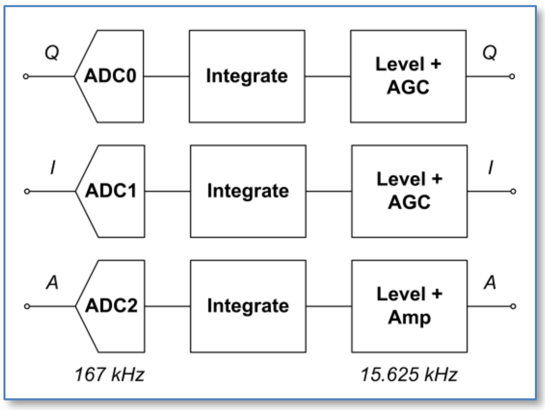
The sampling process uses the ADCs in free-running round robin fashion for all three channels, storing the samples in dedicated buffers in the ADC-FIFO IRQ routine. Each conversion takes 2µsec, after 3 conversions the ADC FIFO generates an IRQ and the ISR stops the conversions.



The image above shows the timing of the ADC-FIFO and Timer IRQ callback routines. The latest samples are taken from the buffers at the beginning of each 64µsec time slot, before starting a new ADC conversion cycle. This cycle could integrate up to 10 samples per channel (i.e. 10x 6µsec) for increased dynamic range.

Sample preprocessing is done every timeslot in the timer callback routine, but both time and frequency domain processing is executed as a background process, which is triggered after the preprocessing. Time domain processing is triggered every slot, while frequency domain processing is only started every 512 slots when a ½ FFT buffer is filled.

3.1.1.1 *Sample preprocessing*



So, the timer callback routine determines the actual sampling rate which is set to 64 µsec (15.625 kHz).

The phase error between the I- and Q-samples is the duration of one ADC conversions; 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.

DC removal

The samples from the ADC are electrically centered on approximately half the reference voltage, which is half the ADC dynamic range (2048). This level is subtracted before integration in the ADC-FIFO callback routine.

A more accurate DC removal process is still recommended, to subtract the low-pass filtered running average signal. This may however take too much overhead and could be omitted.

```
	dc += (sample - dc)/128
```

The RC time is 127*64µsec ≈ 8msec or 125Hz 1st order low pass.

Note that in this scheme only samples that are larger than dc ± 128 will actually contribute to a correction, so prescaling is required to prevent this.

AGC

A signal level estimate is maintained by low-pass filtering the absolute value of the DC-corrected samples. The I and Q streams are amplified with the **AGC gain** factor, which is derived from the low pass filtered ADC level of these I and Q channels. The AGC is just a multiplication factor for the samples, to stretch them to the desired range.

Note that this range is different for TD and FD processing.

VOX

The voice activated switch (VOX) procedure 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 audio level is determined in the Timer callback routine, but the actual VOX detector is implemented in the DSP main loop.

3.1.1.2 *Sample output*

The output of samples to either the audio interface in RX mode or the QSE in TX mode is also handled by the timer callback routine, just before invoking the RX or TX signal processing. The signal is range checked and either clipped or attenuated before sending it to the DAC.

3.1.1.3 *Invoking the signal processing*

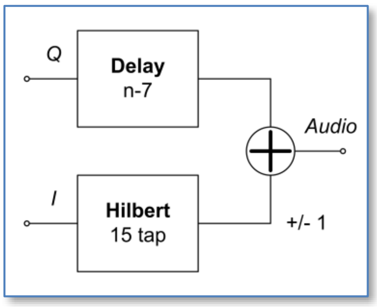
For the TD processing the effective sample rate is also the rate for the RX/TX signal processing routines. For the FD processing the RX/TX functions are invoked only every half FFT-size (1024/2) samples, i.e. when a buffer is filled. This yields a much lower call rate of 305Hz (1/32.768msec), but this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

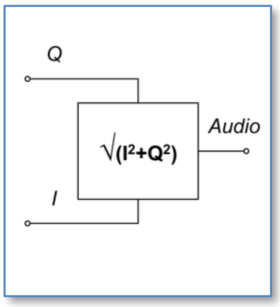
3.1.2 Time domain signal processing (dsp_tim.c)

3.1.2.1 RX stream

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

Demodulation



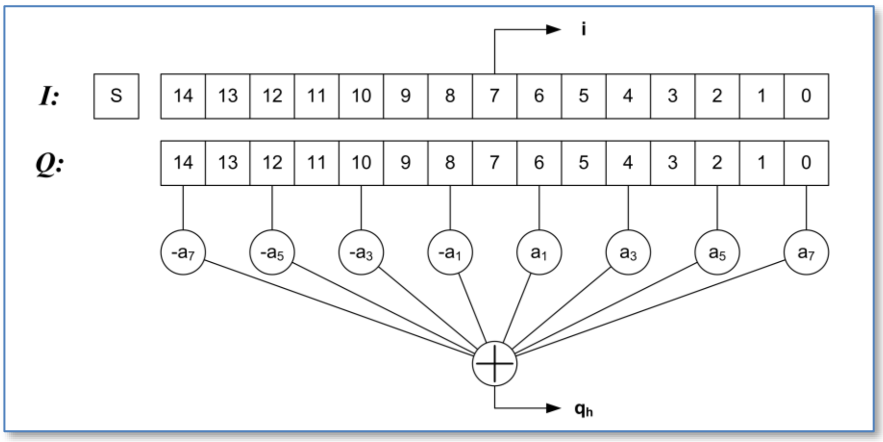


For SSB demodulation, the incoming I and Q samples are stored in a 15-sample delay line. A 15-tap Hilbert transform on the Q channel is done and the result is subtracted from or added to the n-7 sample in the I delay line to obtain the USB or LSB audio output respectively.

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

The resulting audio sample stream is handed to the Audio generation process.

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:

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

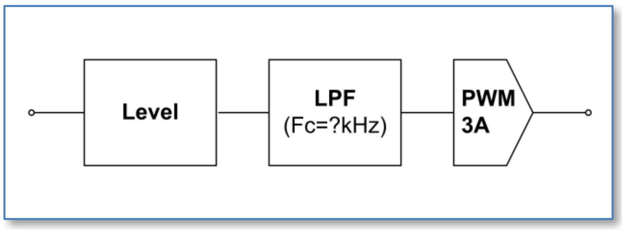
In this function $w(n)$ is a windowing function, for example a Hamming (raised cosine) window:

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

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.

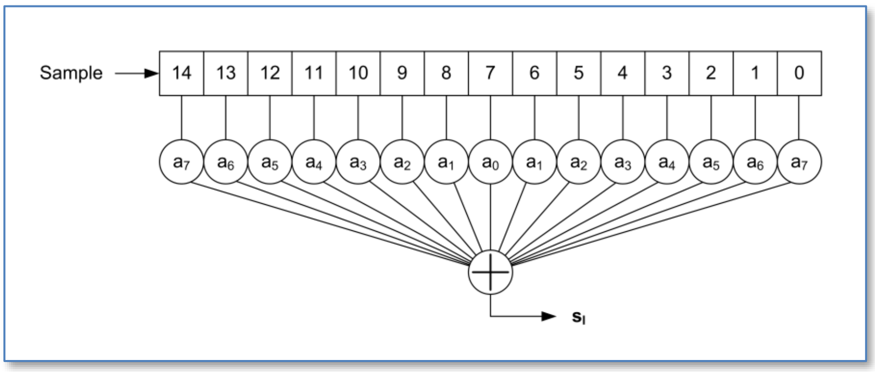
-7	-5	-3	-1	1	3	5	7
-0.00728	-0.03224	-0.13631	-0.60762	0.60762	0.13631	0.03224	0.00728

Audio generation



The demodulated audio samples pass through a level detector, which generates an AGC feedback signal in order to scale the output to within the PWM/DAC range. The scaling is logarithmic, in factors of 2 (6dB steps), enabling simple bit shifts to be used as amplification/attenuation in the sampling block.

Low Pass Filters



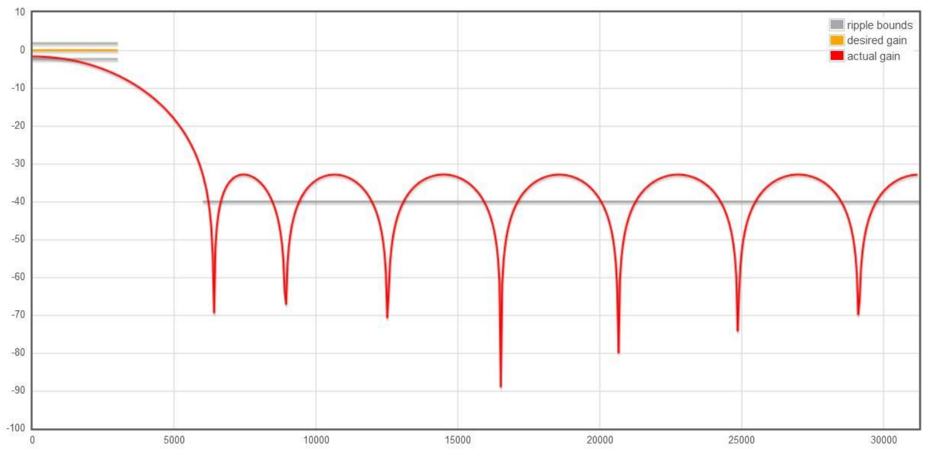
Several 15-tap low pass filters have been created, with a corner frequency of F_c=3kHz. The stop-band depends on the actual sample rate the filter is designed for. It usually starts around 5kHz, with a level of -40dB or better. The code contains filters for 62.5 kHz, 31.25 kHz and 15.625 kHz sample rates,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

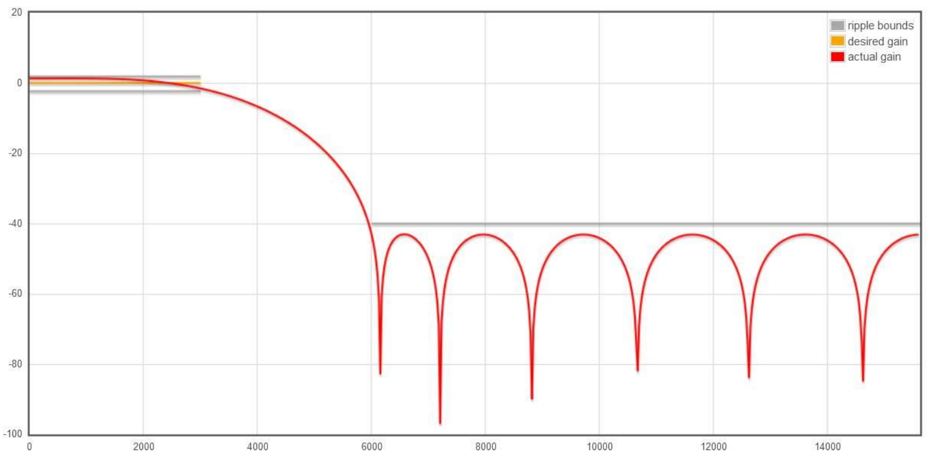
These low pass filters are simple symmetric FIR filters, that represent the impulse response of the desired low pass behavior. They consist of 15 signed integer arrays. Hence, per sample 15 multiplications and additions need to be done, but the RP2040 has a single cycle 32bit MPY instruction, so that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

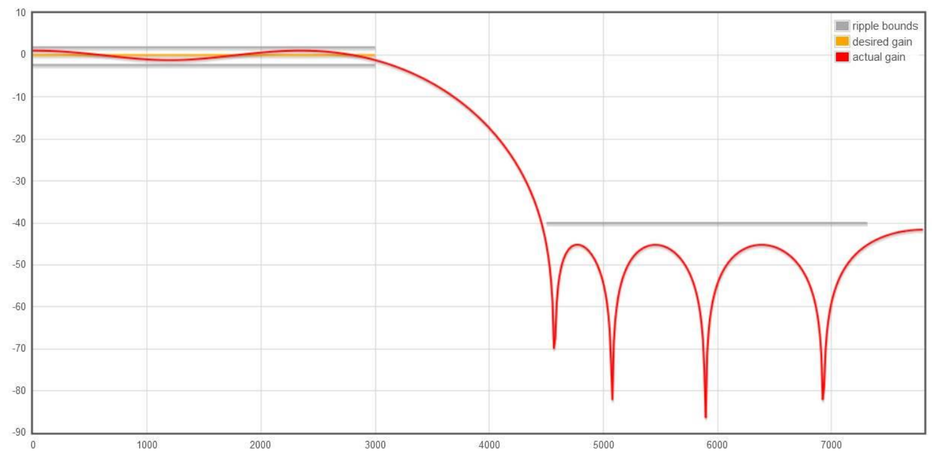
To find the proper coefficients, see for example Iowa Hills DSP tools or the T-Filter on-line calculator:

- 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:

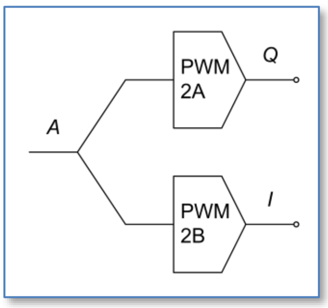


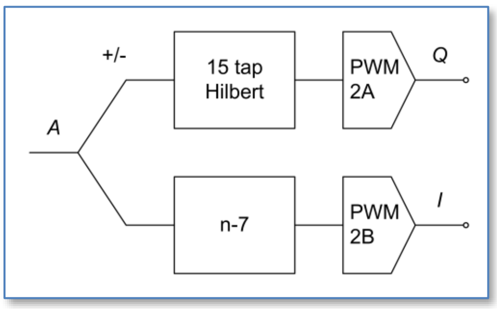




Note that for the 3kHz low pass filter, a 15 tap FIR algorithm works best at lower sample rates. For high sample rates it would be better to use more taps. Currently only the low sample rate filter is used in uSDR-Pico.

3.1.2.2 *TX stream*





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.

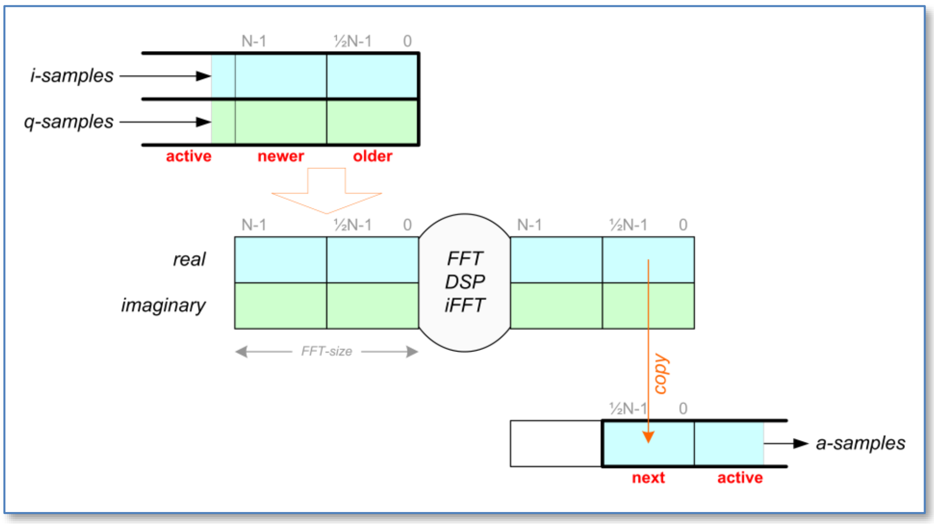
The dynamic range of the sample streams is matched with that of the DAC range, before output.

3.1.3 Frequency domain signal processing (dsp_fft.c)

While Time Domain signal processing is done for every incoming sample, 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 signalled only every 512 samples (512x64usec ≈ 32msec), and runs in the background while interrupted by the timer at raw sampling rate.

3.1.3.1 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.



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.

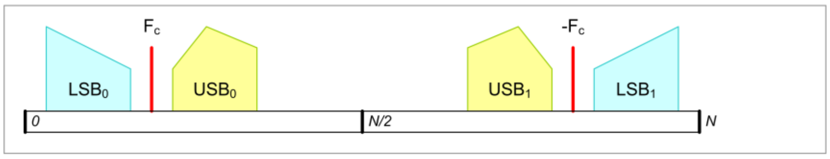
3.1.3.2 Signal processing

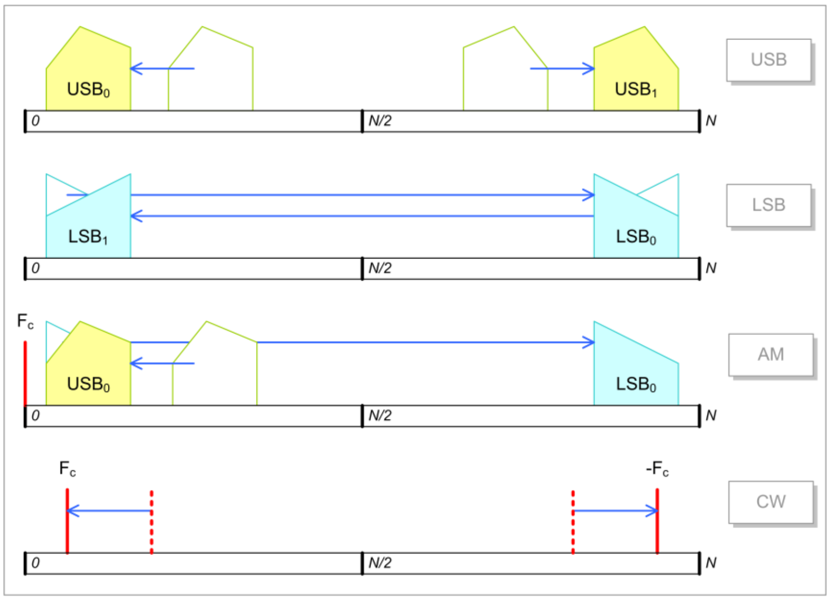
The signal processing engine follows the sequence of transformation to the frequency domain, applying the band filtering and shifting as well as transformation back to the time domain. The samples in the audio domain are real, so a conversion from (and to) the complex representation is required.

RX case:

<<Shift & Filter>>

To enable the filtering, the carrier frequency must not be downconverted to 0 Hz, but rather to somewhere in the center of the frequency band resulting from the FFT. With a sampling rate of 15.625kHz, the *offset* frequency F_c should be somewhere around 3.9kHz. Depending on the desired modulation mode, different ranges with respect to this *offset* are filtered out, and shifted to the proper place in the spectrum buffer. For example (only real spectum shown):





For AM the upper and lower sidebands contain the same information, i.e. the spectrum about the Fc is symmetric. This implies that the corresponding sidebands could be mirrored and added for a 3dB gain.

For CW the filter can be narrow around F_c and the effective shift should be reduced with the desired tone (e.g. 900Hz). After the iFFT of this filtered spectrum, the real part of the complex time samples are copied to the audio samples buffer.

<u>TX case:</u>

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

3.1.4 Fast Fourier Transform (fix_fft.c)

The crux 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.

3.1.4.1 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.

3.1.4.2 FFT implementation

The FFT algorithm is explained in more detail in an appendix.

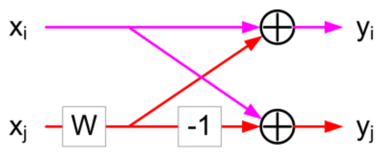
The first stage of the FFT algorithm is the reordering of the time domain samples; to be precise, the samples with indexes that are bit-reverse of each other need to be swapped. The array size is 1024 samples, so the index is 10 bits. The swap is then for example between samples [1111000001_b] and [1000001111_b]. 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.

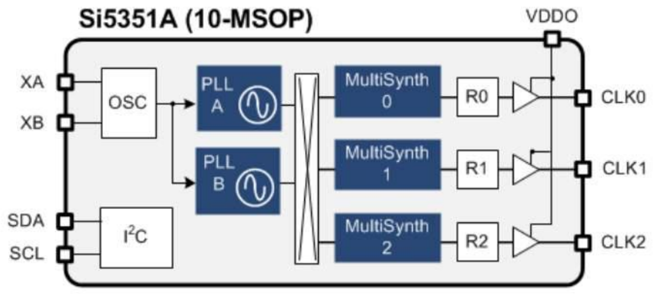
The second stage consists of a number of nested loops, calculating and adding the butterflies. One such butterfly can be represented as follows:



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.

3.2 Quadrature VFO (si5351.c)



The Si5351A is a triple clock generator, that can be controlled through I^{2}C 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 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 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

3.4 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.

3.5 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.

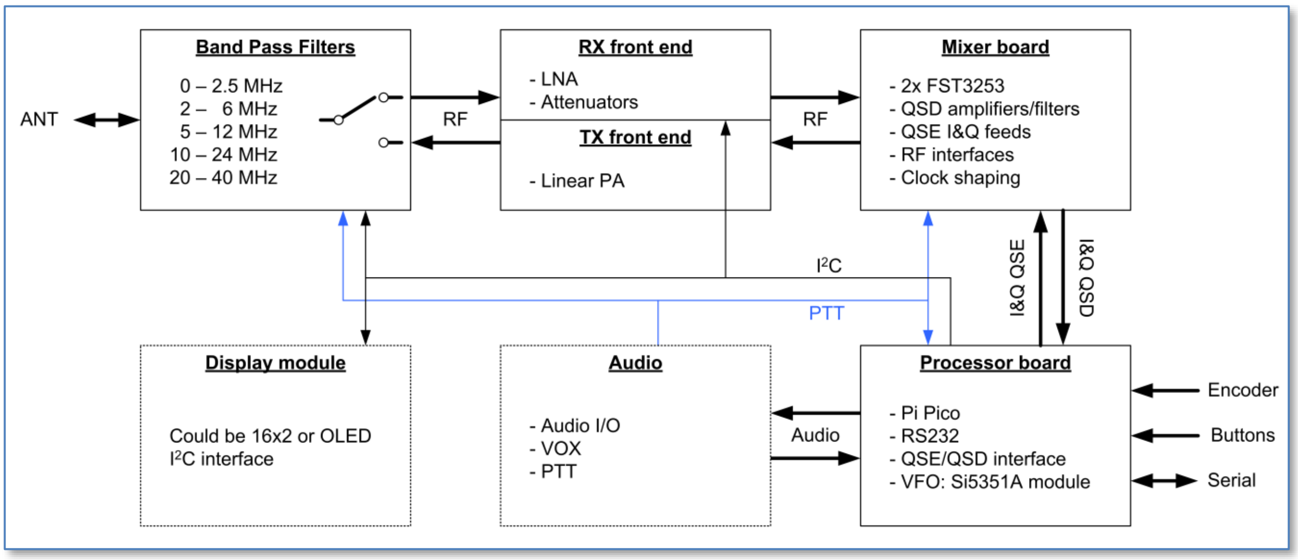
The shell vocabulary is contained in an array of strings. Whenever a CR/LF is entered on stdin, the collected characters are treated as command-line, and a match is attempted with the strings in the shell array. When a match is found, the corresponding handler is invoked, with the remainder of the command-line.

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

4 Hardware prototype

The uSDR prototype is based on off the shelf modules and is installed in a Teco 1500 enclosure. It mainly serves as a test and experimentation environment, possible basis for a more integrated implementation on the longer term. However, the modularity of the prototype is good for experimentation, since it allows to swap out certain functions of the system that do not function appropriately. The main part of the functionality is realized in software.

The modular architecture is set up as follows:



The signal path leads from Audio board to BPF and vice versa. Internal control is done through an I^{2}C 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.

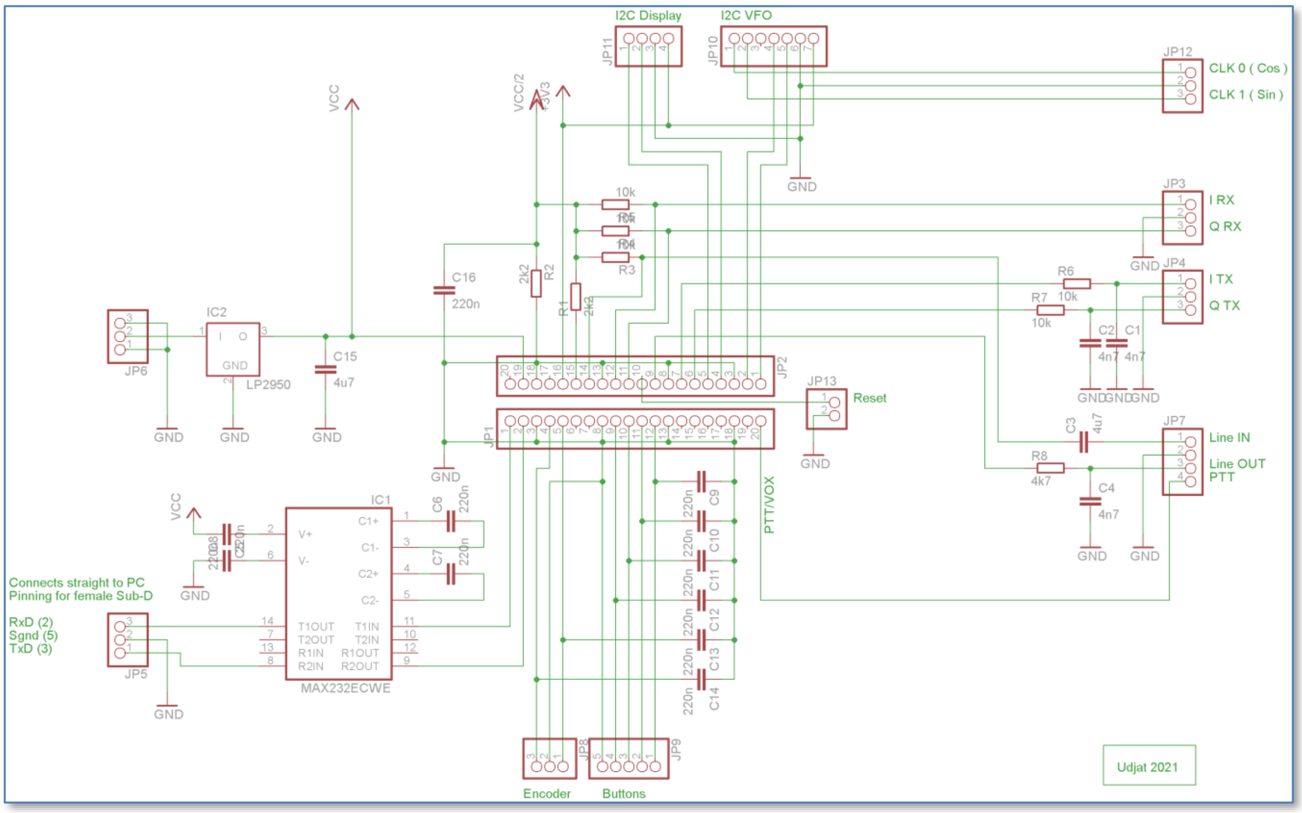
In summary, the modules are:

- Processor board: This is simply a carrier for the Pi Pico as well as the VFO module. The interfacing comprises the RS232 levelling, switch debouncing and PWM/DAC filtering
- Audio board: This contains all analogue audio handling, VOX and external PTT interfaces.
- Display module: The display module can be anything suitable with an I^{2}C interface. The SW is made for a regular 16x2 alphanumeric type.
- Mixer board: This is a design based on two FST3253 multiplexers. Clock shaping is done with a 7400 ACT or HCT, and analogue IF signal handling with three LM4562.
- RX/TX front ends: These two boards contain the TX PA and the RX LNA and attenuators.
- Filter board: This contains a set of 5 switched bandpass filters, that go in between RF front-end and antenna. The switching between the RX and TX path is done by a PTT controlled relay.

The Processor, Mixer, RX and TX boards all have the same form factor, 2” x 3.2” (51x82mm). The Filter board did not fit and is slightly bigger (51x94mm).

It could be preferrable to move the TX-PA stage between antenna and BPF. This would however require another set of low-pass filters. The reason is to filter unwanted signals that originate in the mixer.

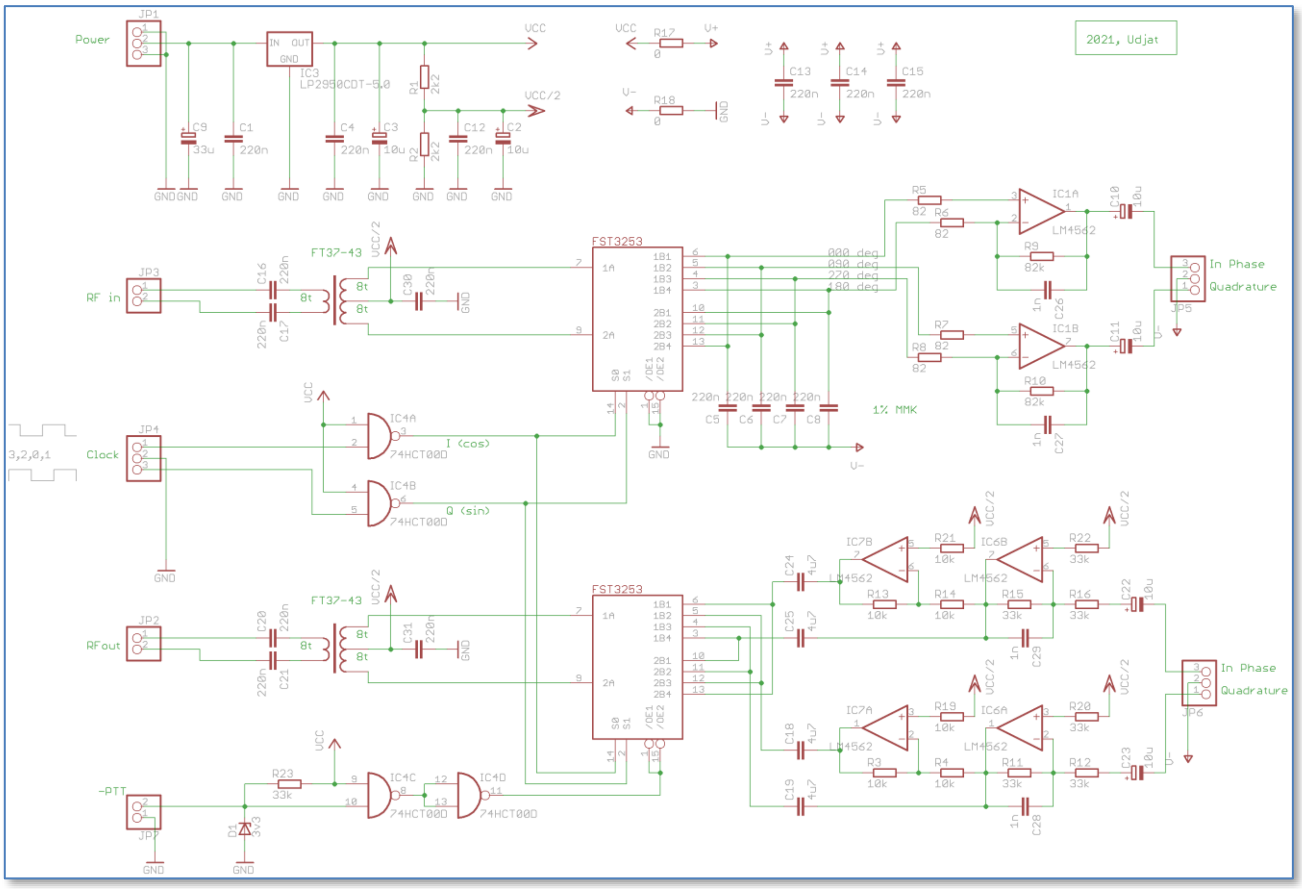
4.1 Processor board



The processor board hosts the Pico module, the Si5351A VFO module and the user interfaces. There is also a serial interface that provides a monitor with a command line interface. The audio handling and PTT/VOX circuitry is moved outside the CPU/IO board, for modularity reasons.

The I and Q output filters have a corner frequency of 3.4kHz, the line output filter 7.2kHz.

4.2 Mixer board

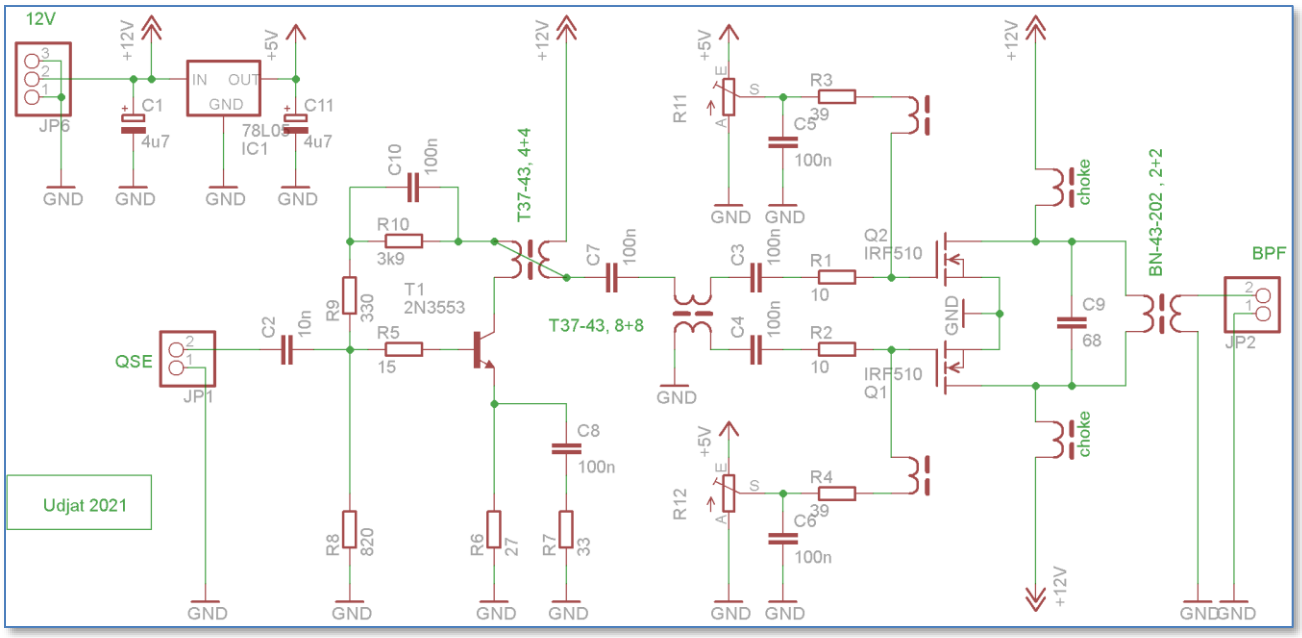


The mixer board hosts the two mixers, QSD for RX branch and a QSE for the TX branch. The switches are clocked from the Si5351 board, through two HCT NAND gates that take care of level shifting and some signal cleanup. TX path can be disabled when PTT is not active. RX path is always enabled.

The resistors of the TX input opamps have been increased in order to obtain a higher input impedance. This makes sure that the signal is not divided too much. The low-pass corner frequency is about 4.8kHz.

A point of attention is the QSE branch, the output signal seems to have a lot of distortion. Maybe a couple of load resistors on the FST3253 outputs or some bias adjustment will do the trick here. Another consideration is to remove the separation capacitors and the bias circuit, since the opamp outputs already have a bias. These then have to be identical in all 4 paths, in order to retain sufficient carrier and opposite sideband suppression.

4.3 TX board

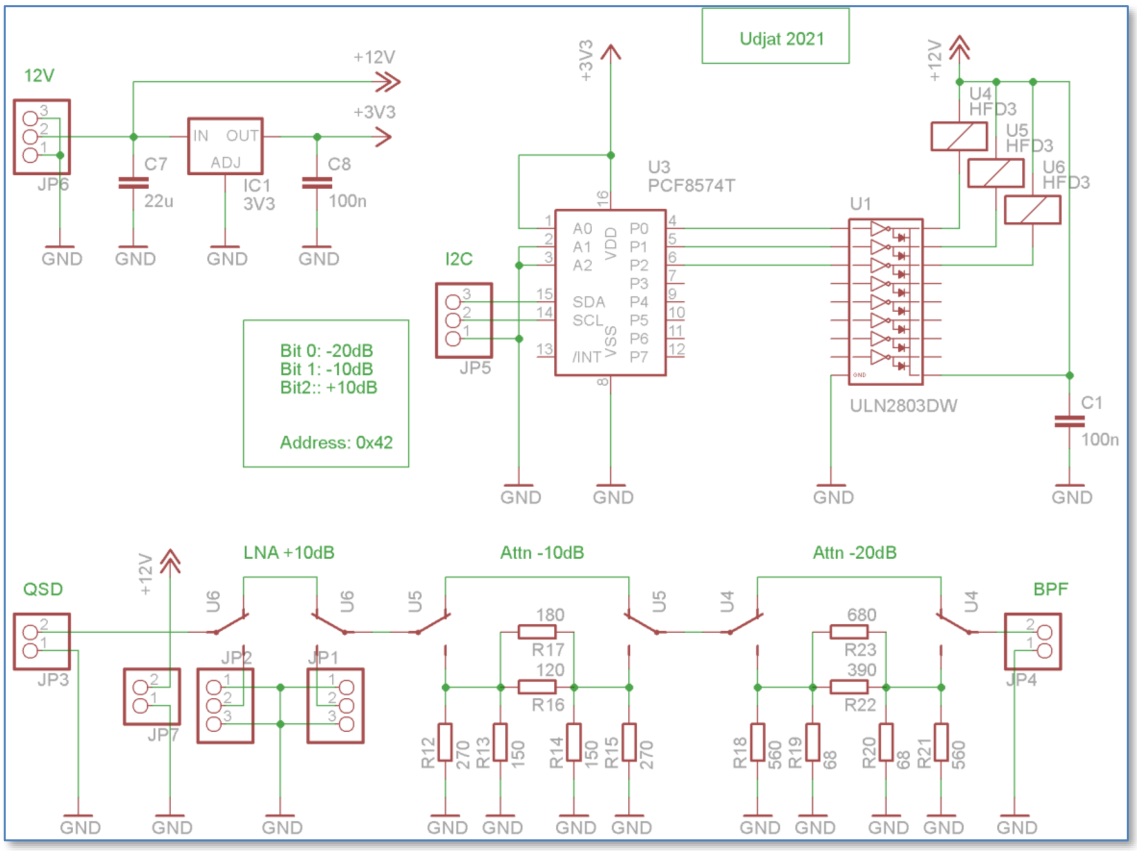


The new proto TX board consists of a class AB, IRF510 based push-pull amplifier, driven by a regular class A stage. This driver has a 2N3553 for proto, but that could be anything like 2N228018, 2N2219, 2N2222 etc.

Bias is adjusted to just above cutoff, so a standing current of about 20mA per MOSFET drain, to be adjusted for signal symmetry when the amplifier is operational.

This is just an example PA, which needs to be further tested.

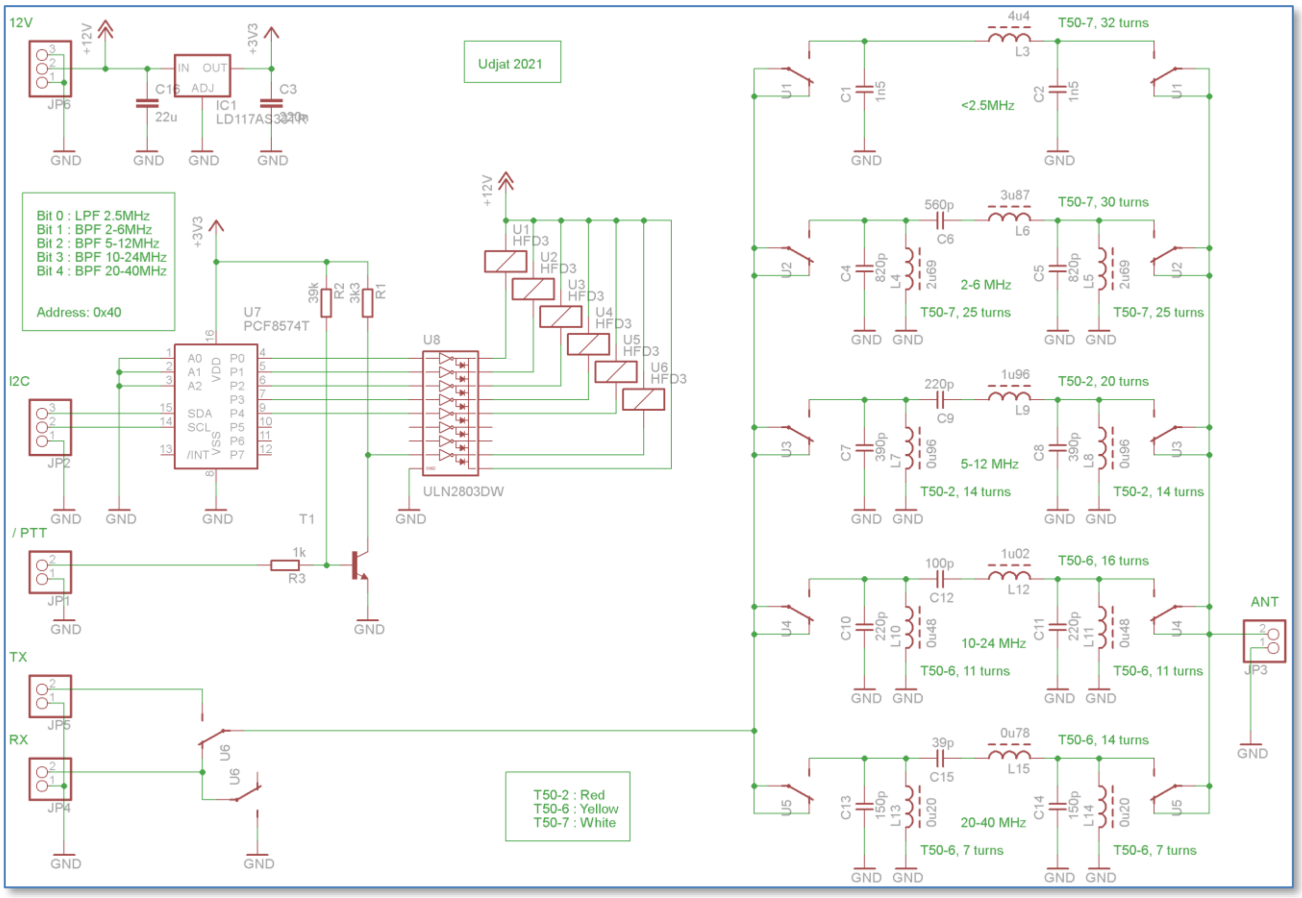
4.4 RX board



The RX board just contains the selectable attenuators and a low noise amplifier.

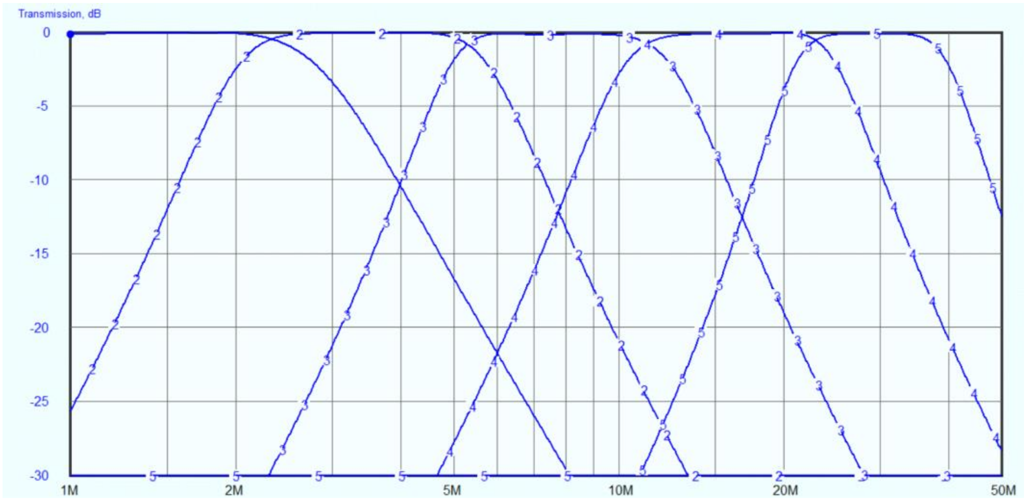
One thing to change in subsequent versions is the interface PCF8574 – ULN2803, the I^{2}C expander can barely provide the drive current for the Darlington array.

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.

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

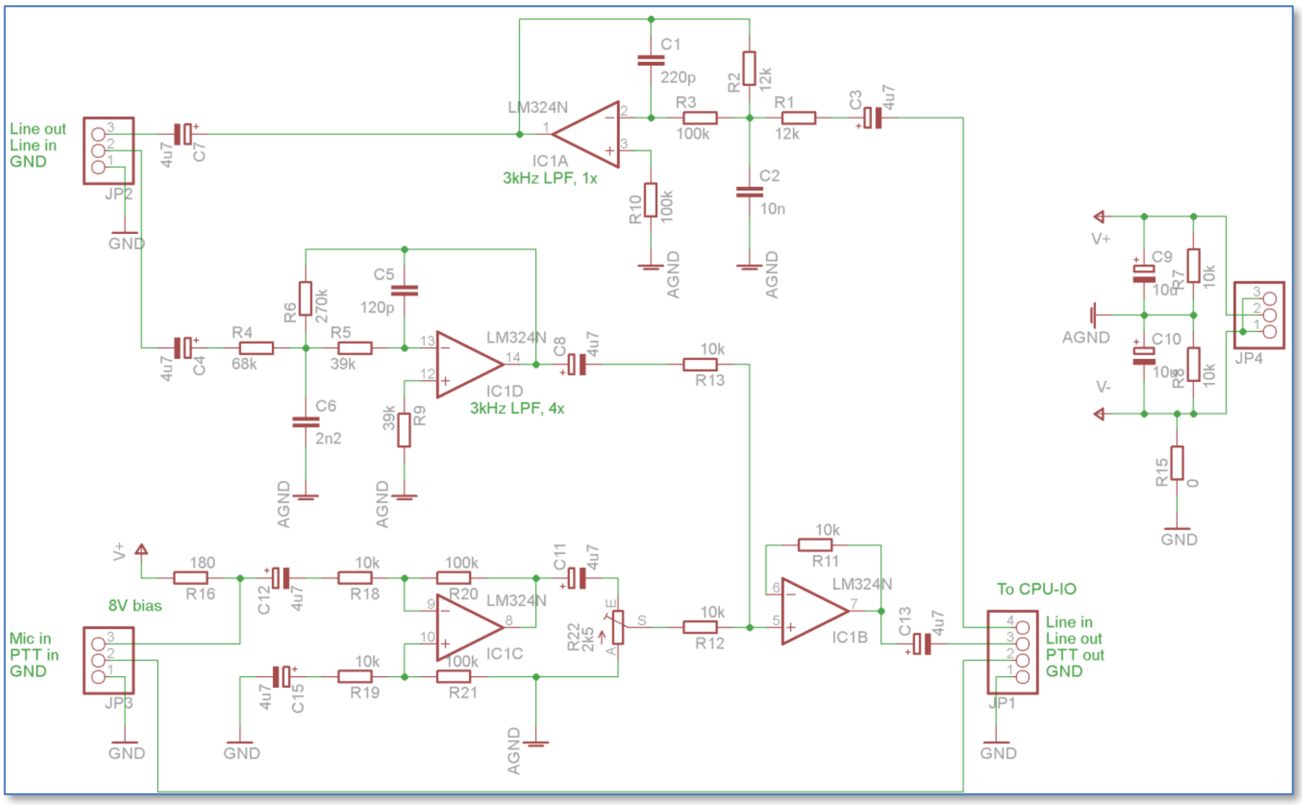


One thing to change in subsequent versions is the interface PCF8574 – ULN2803, the I^{2}C expander can barely provide the drive current for the Darlington array.

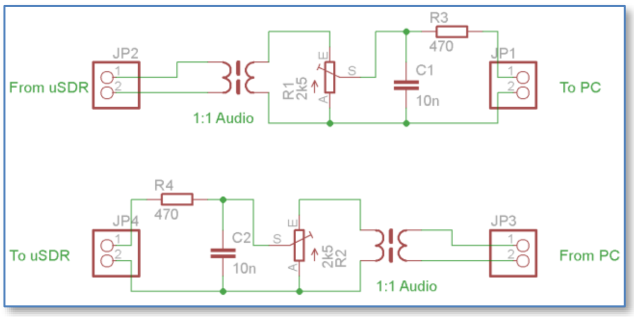
Another slight modification is R3 in the PTT circuit, which was needed because the basis of T1 pulled the PTT signal too far down.

4.6 Audio module

Two modules were made on proto boards to implement the audio handling.

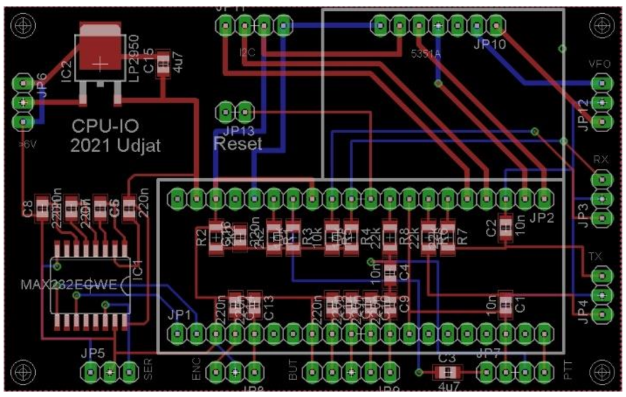


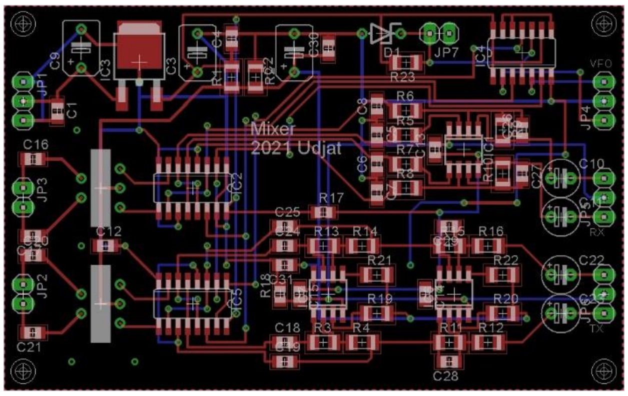
The first is buffering and pre-amplifying the line and microphone channels, as well as an additional 3kHz low pass filter. The line in signal is amplified 6dB to get some extra range for level optimization. The signal that goes to the ADC should be close to $3V_{pp}$. The mike signal is more or less balanced and amplified 10dB. In the current version this is still to be tested. The mike PTT signal is only enabled when No VOX is selected, otherwise the PTT is level controlled.

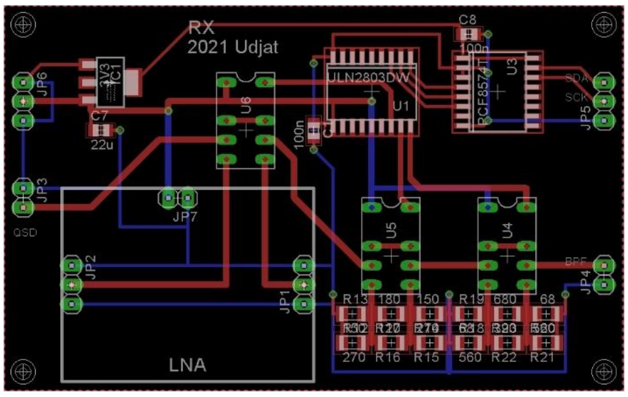


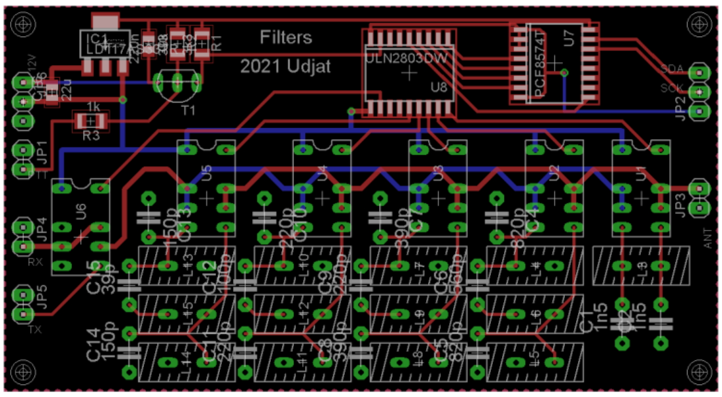
The second circuit is used in the cable that connects PC with uSDR-Pico, and serves as galvanic separation; PCs can be rather noisy… The trimmers can be used to adjust the required levels.

4.7 PCB layout





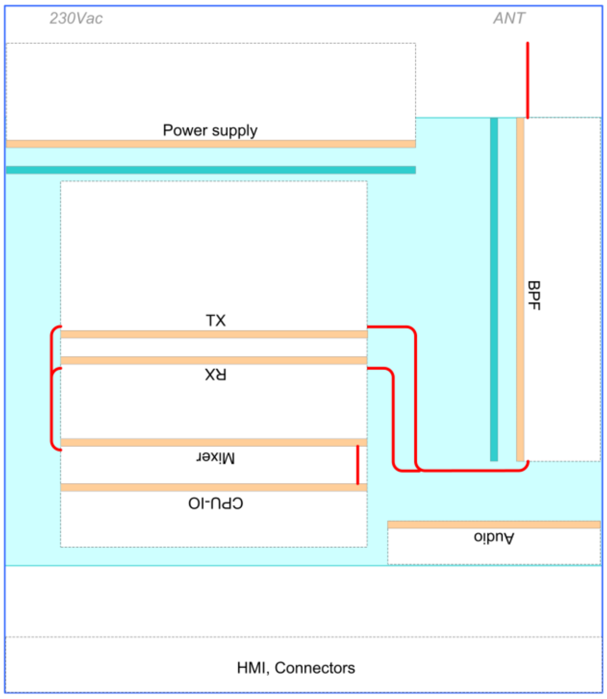


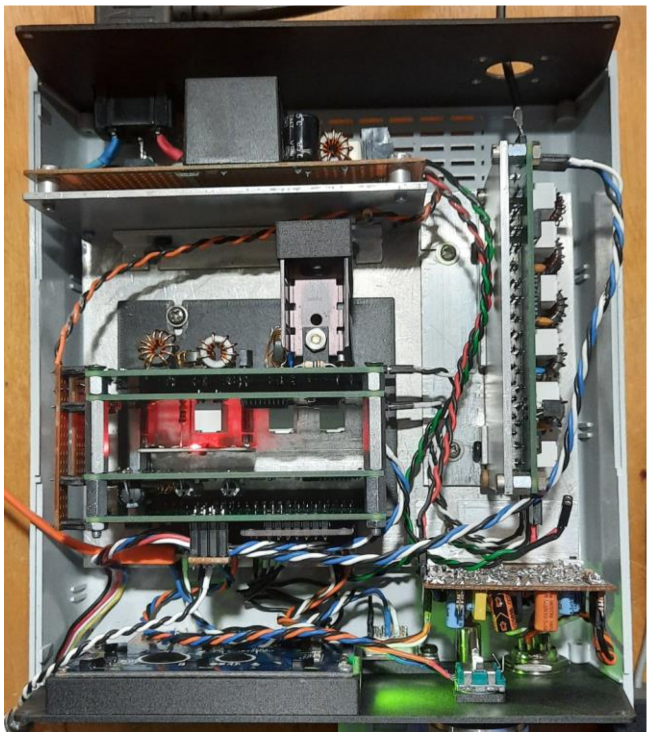


4.8 Mechanics

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:

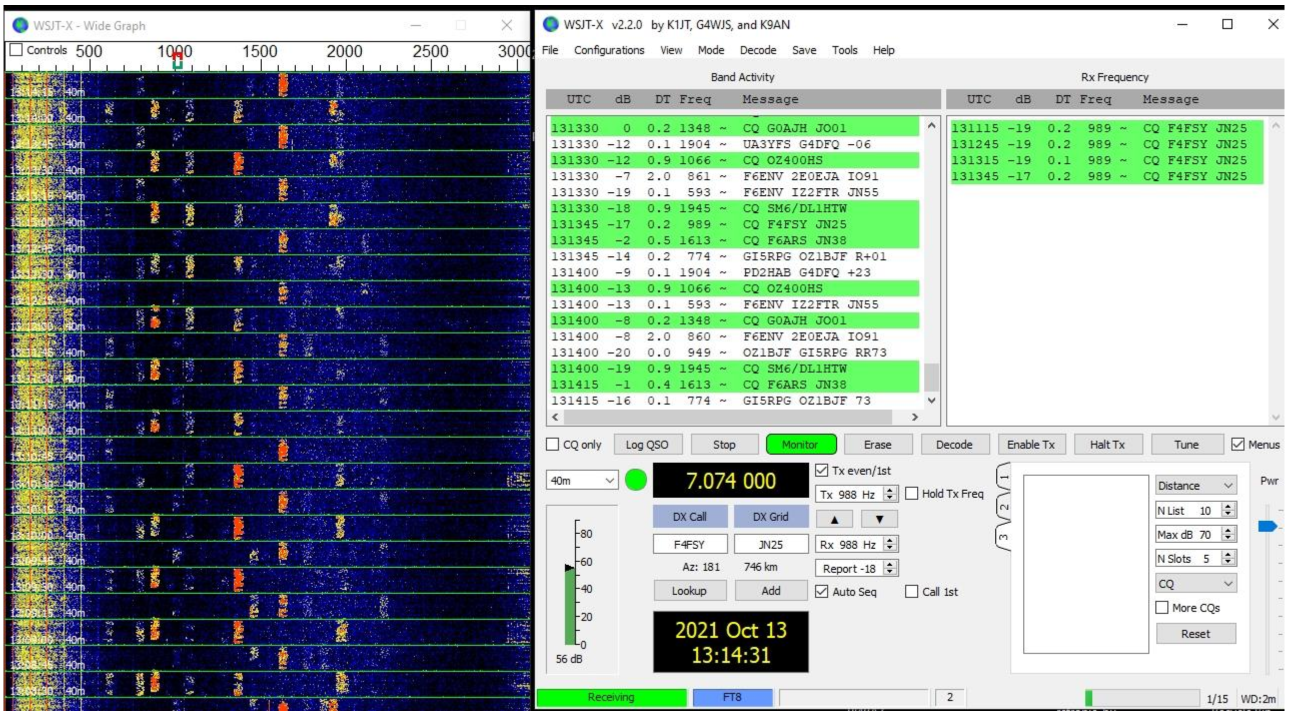




The grey parts represent aluminum supports, which carry the various PCBs shown in tan color. Component side is where the name tag is. Dashed white areas indicate approximate space taken by the components.

5 Testing

The mixer, processor and RX frontend work fine, at least they are a sufficient basis for further software development. BPF and RX/TX frontend are next to be prototyped on a PCB. As of now, the plan is to take the audio processing off the Processor board, and make a separate module that contains the PTT, the audio input amplifier, the VOX circuits and an audio output buffer.



Band Activity

```
  UTC    dB   DT Freq    Message
131330    0  0.2 1348 ~  CQ G0AJH JO01
131330 -12  0.1 1904 ~  UA3YFS G4DFQ -06
131330 -12  0.9 1066 ~  CQ OZ400HS
131330  -7  2.0  861 ~  F6ENV 2E0EJA IO91
131330 -19  0.1  593 ~  F6ENV IZ2FTR JN55
131330 -18  0.9 1945 ~  CQ SM6/DL1HTW
131345 -17  0.2  989 ~  CQ F4FSY JN25
131345  -2  0.5 1613 ~  CQ F6ARS JN38
131345 -14  0.2  774 ~  GI5RPG OZ1BJF R+01
131400  -9  0.1 1904 ~  PD2HAB G4DFQ +23
131400 -13  0.9 1066 ~  CQ OZ400HS
131400 -13  0.1  593 ~  F6ENV IZ2FTR JN55
131400  -8  0.2 1348 ~  CQ G0AJH JO01
131400  -8  2.0  860 ~  F6ENV 2E0EJA IO91
131400 -20  0.0  949 ~  OZ1BJF GI5RPG RR73
131400 -19  0.9 1945 ~  CQ SM6/DL1HTW
131415  -1  0.4 1613 ~  CQ F6ARS JN38
131415 -16  0.1  774 ~  GI5RPG OZ1BJF 73
```

Rx Frequency

```
  UTC    dB   DT Freq    Message
131115 -19  0.2  989 ~  CQ F4FSY JN25
131245 -19  0.2  989 ~  CQ F4FSY JN25
131315 -19  0.1  989 ~  CQ F4FSY JN25
131345 -17  0.2  989 ~  CQ F4FSY JN25
```

```
40m        7.074 000      [✓] Tx even/1st
                          Tx 988 Hz
DX Call    DX Grid
F4FSY      JN25           Rx 988 Hz
Az: 181    746 km         Report -18
                          [✓] Auto Seq
56 dB      2021 Oct 13
           13:14:31
Distance   N List 10   Max dB 70   N Slots 5   CQ
Receiving   FT8   2   1/15   WD:2m
```

As the screen-dump shows, the current version works nicely on the 40m FT8 channel. The AGC now also works, albeit not yet controllable from the menu. It makes sure that the output signal is maintained between 1.6V_{pp} and 3.3V_{pp}.

In the V2.0 the menus for band filter selection, pre-amp or attenuator, modulation mode and AGC are working, as well as a simple command shell on the serial port. This port needs to be connected to a PC with a straight RS232 cable.

The QSE side seems a bit more complicated. An upper sideband modulation of a single tone should give a constant carrier frequency up-shifted by the tone frequency. This is a nice way to check whether the signs in the modulation chain are in order (see dsp.c). Another test is with two tones, which should show up as a carrier that is modulated in amplitude with the frequency difference between tones. The carrier and opposite sideband should be suppressed though, hence the signal is not exactly AM.

The output from the mixer can be tested with this background information. The circuit as presented above seems to have some issues, although a single tone and also FT8 can be detected.

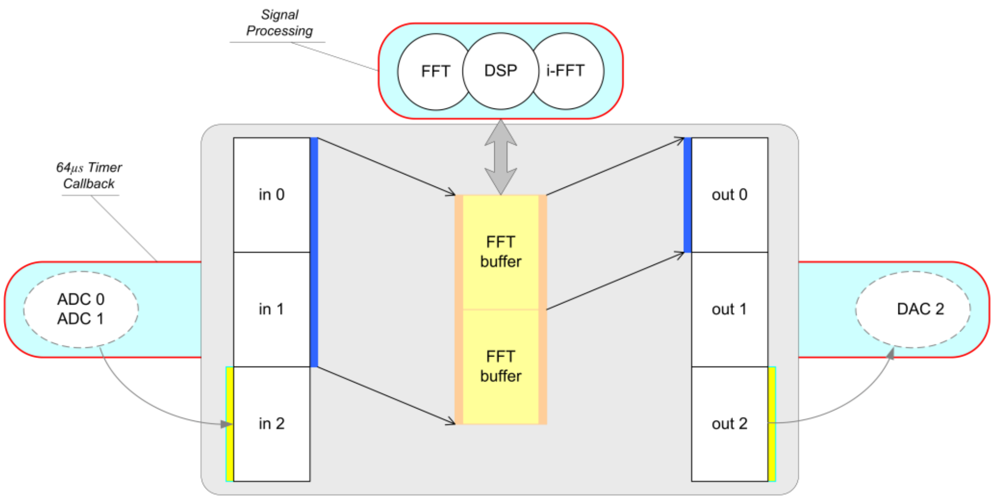
Annex A: Frequency domain processing

5.1 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.

5.2 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.

5.3 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:

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

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·**^{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***.

5.4 FFT implementation

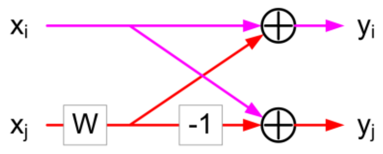
The first stage of the FFT algorithm is the reordering of the samples; to be precise, the samples with indexes that are bit-reverse of each other need to be swapped. The array size is 1024 samples, so the index is 10 bits. The swap is then for example between samples [1111000001_b] and [1000001111_b]. 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”.

5.5 Sequencing

5.5.1 *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.

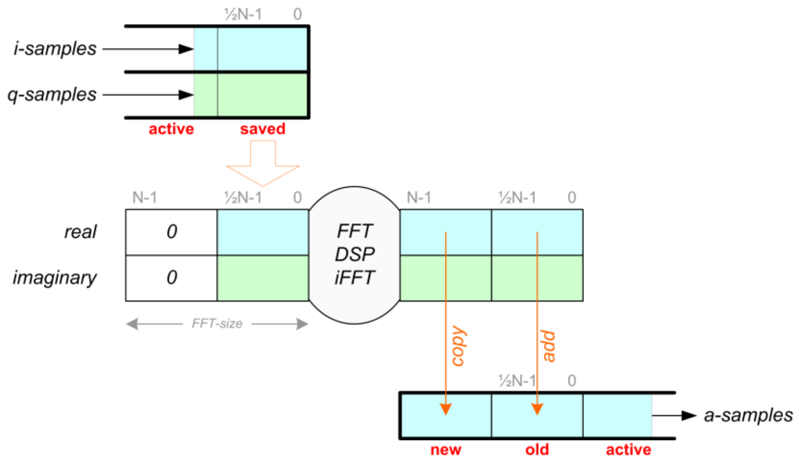
The signal should be amplified to fill the FFT dynamic range, i.e. up to about the range of `int16_t`. To do this, an average signal strength indicator is calculated so that a multiplier (call it an AGC) can be deduced. Normally the input signal is less than $1V_{pp}$, which is 1/3 of the ADC range of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

5.5.2 *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.

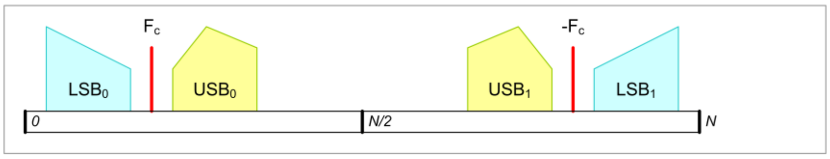
5.5.3 *Signal processing*

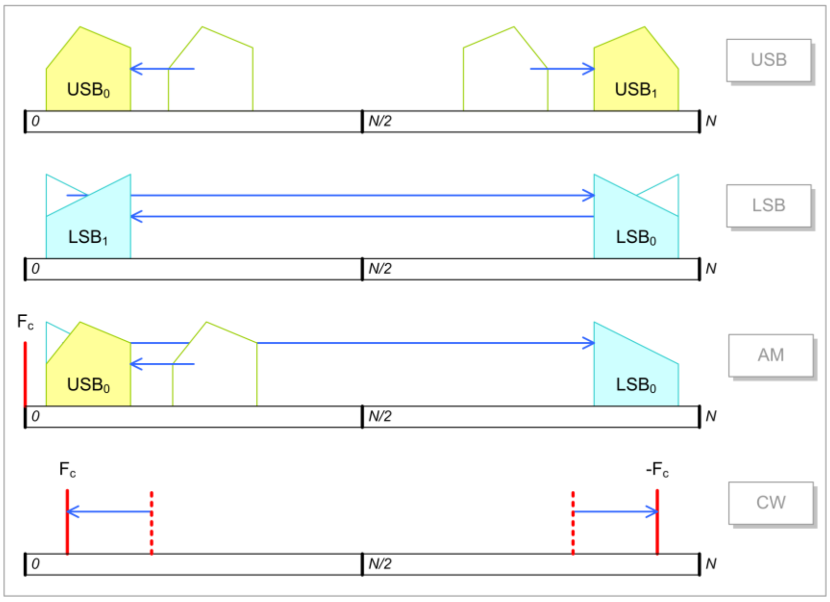
The signal processing follows the sequence of transformation to the frequency domain, applying the filtering/shifting and transformation back to the time domain. The samples in the audio domain are real, so a conversion from (and to) the complex representation is required.

RX case:

<<Shift & Filter>>

To enable the filtering, the carrier frequency must not be downconverted to 0 Hz, but rather to somewhere in the center of the frequency band resulting from the FFT. With a sampling rate of 15.625kHz, the *offset* frequency F$_c$ should be somewhere around 3.9kHz. Depending on the desired modulation mode, different ranges with respect to this *offset* are filtered out, and shifted to the proper place in the spectrum buffer. For example (only real spectrum shown):





For AM the upper and lower sidebands contain the same information, i.e. the spectrum about the Fc is symmetric. This implies that the corresponding sidebands could be mirrored and added for a 3dB gain.

For CW the filter can be narrow around F$_c$ and the effective shift should be reduced with the desired tone (e.g. 900Hz). After the iFFT of this filtered spectrum, the real part of the complex time samples are copied to the audio samples buffer.

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

Overlap-Add method

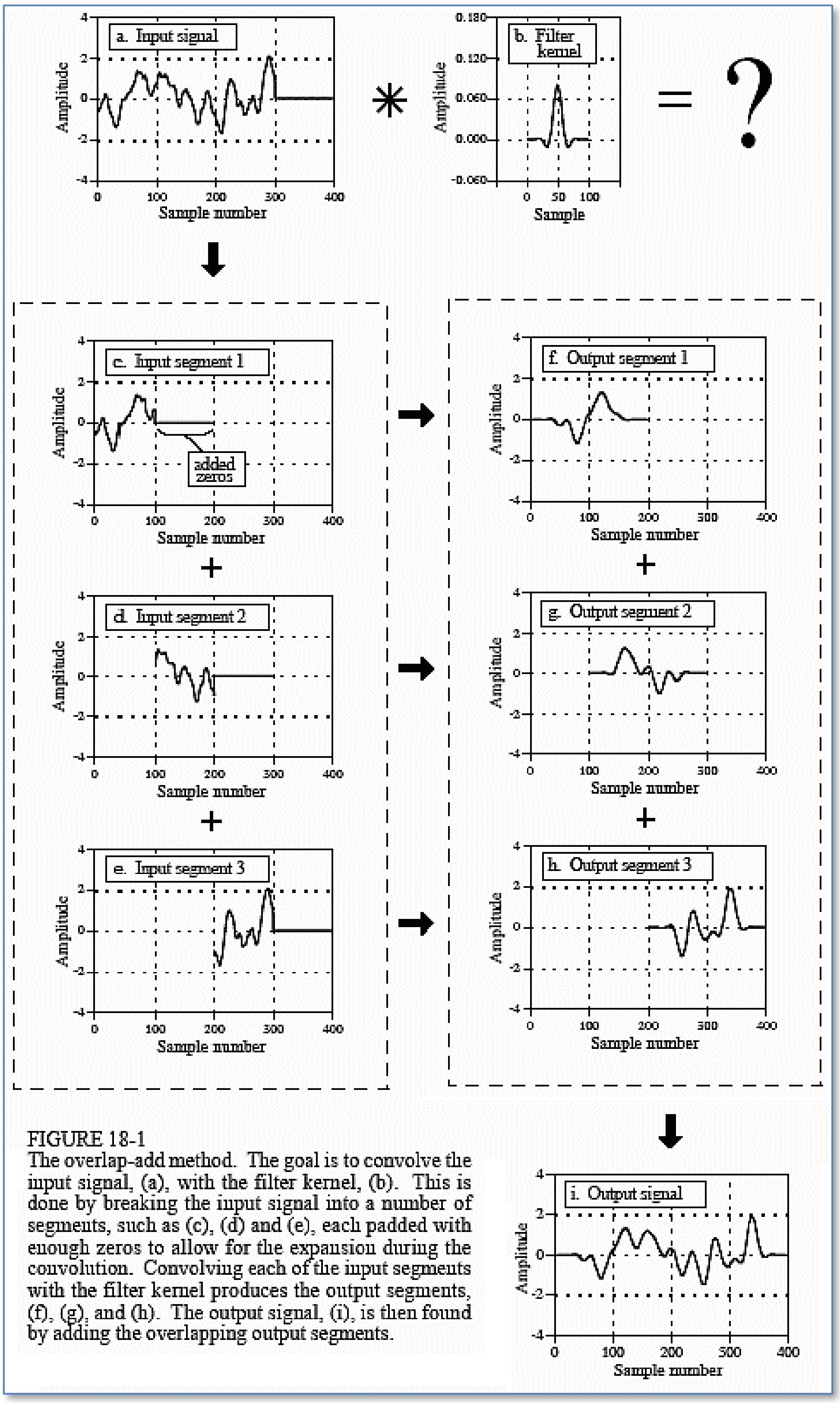


FIGURE 18-1
The overlap-add method. The goal is to convolve the input signal, (a), with the filter kernel, (b). This is done by breaking the input signal into a number of segments, such as (c), (d) and (e), each padded with enough zeros to allow for the expansion during the convolution. Convolving each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