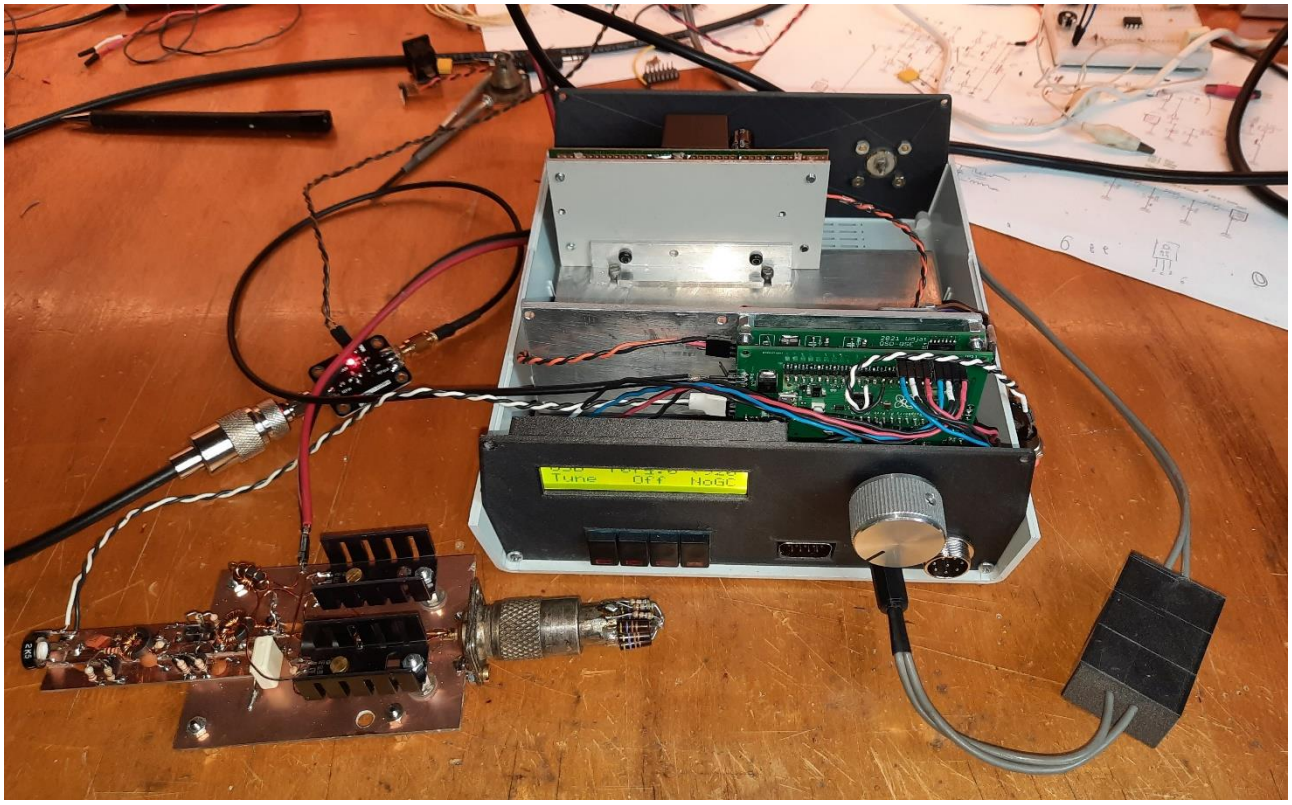
Micro SDR on a Pi-Pico

Arjan te Marvelde, October 2021 (initial version May 2021)



Since 2020, Raspberry offers a new board 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 is not fully mature yet (May 2021) but at least it provides a quick start in actually setting this **Pi-Pico** to use.

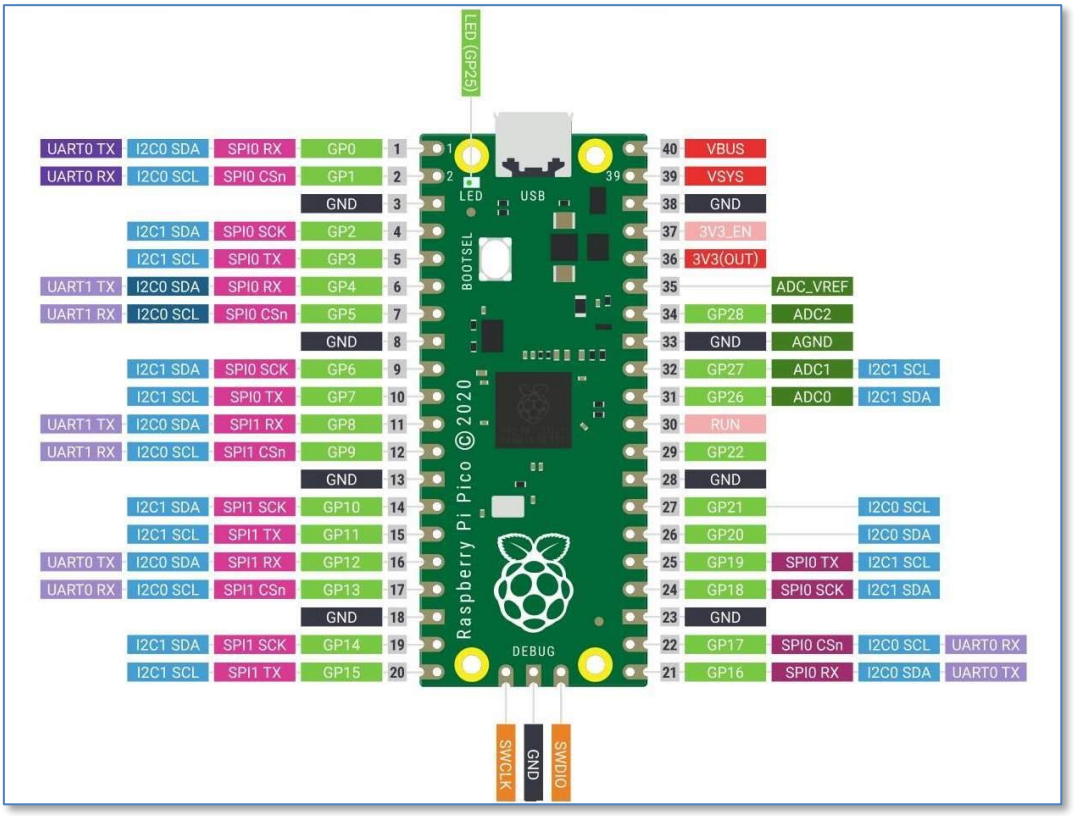
This document describes a test implementation of a small SDR, inspired by Hans Summers' **QCX** and everything that followed after that, such as **µSDX**. Main deviation in hardware is the use of FST3253 based mixers for both RX and TX paths, which makes modularization and experimentation a bit easier.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

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 Teco enclosure.

This project is highly experimental and remains a work in progress, I spend some time on it every now and then.
The project is maintained on GitHUB: https://github.com/ArjanteMarvelde/uSDR-pico.

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

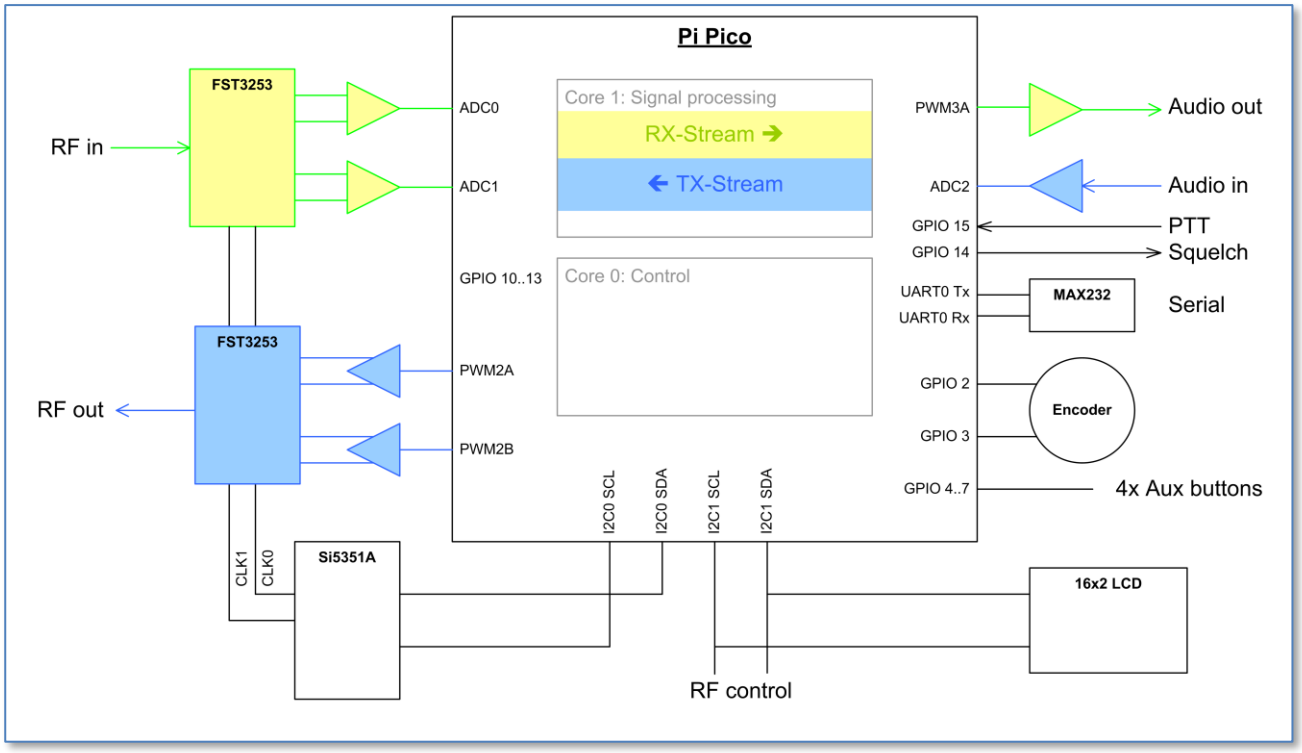
UART0 Tx	GP0	Vbus		Not used
UART0 Rx	GP1	Vsys		5V power input
	GND	GND		
Encoder A input	GP2	3V3 en		Not used
Encoder B input	GP3	3V3 out		3V3 to peripherals
Not used	GP4	Va ref		ADC 3V3 reference output
Not used	GP5	GP28		ADC2: Audio input
	GND	GND		
Aux button 1 input	GP6	GP27		ADC1: QSD Q-channel 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		
	GP10	GP21		PWM 2B: QSE Q-channel output
	GP11	GP20		PWM 2A: QSE: I-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

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 DACs have a maximum range of 3V3.

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



Code layout

The processor has two cores that work independently to a large extent, 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) 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 smallish cache-memory part of the SRAM. Normally this is okay, but time-critical portions of the code can also be loaded in SRAM at boot time. Also, the time critical stuff is located in *core1*, and this should be given priority over *core0* for resource arbitration.

The signal processing on *core1* is split up in two streams, an RX and a TX stream. All three ADC inputs are sampled continuously on highest speed in Round-Robin fashion, and the interrupt handler merely copies the most recent conversion results into buffers. This way, ADC samples for every channel are taken every 6us. The RX stream takes the latest QSD I and Q samples from the buffer, processes these and outputs samples through a PWM-based DAC to the audio interface. This introduces a bit of jitter between I and Q channels, but this is considered negligible compared to the demodulated audio signal bandwidth. 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.

Each stream runs on an 16usec basis (62.5 kHz) which is controlled by a timer running on *core0*. The timer callback just pushes a mode word (being TX or RX) through the inter-core FIFO, which subsequently sets off the associated process in *core1* (so either TX or RX processing, or possibly even both).

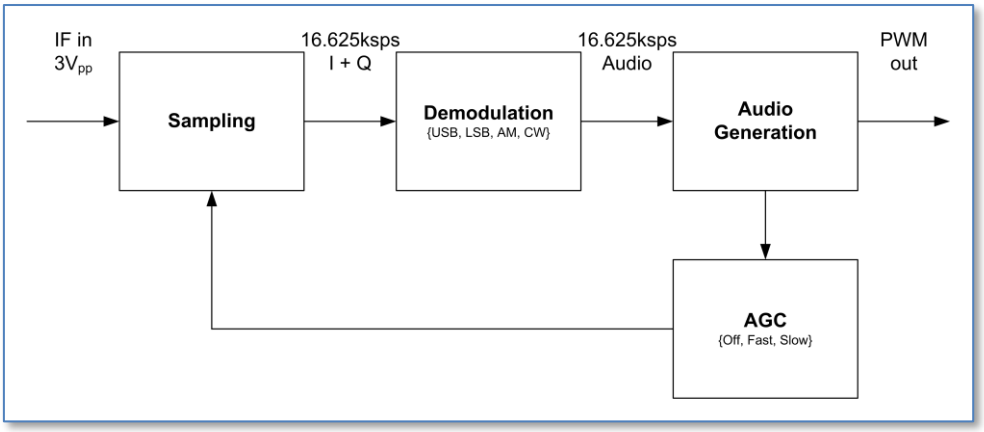
The Pico has two I^2C buses, so we can make it easy and connect one dedicated to the Si5351 and use the other for the remaining devices. There are four GPIOs reserved for auxiliary buttons. These could be used as direct controls, or with a binary decoder to increase the number of control lines.

Overview of the files:

uSDR.c	The main loop and system initialization. The main loop further takes care of all user interfacing.
dsp.c	All signal processing, TX and RX branches, and sample timing.
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.

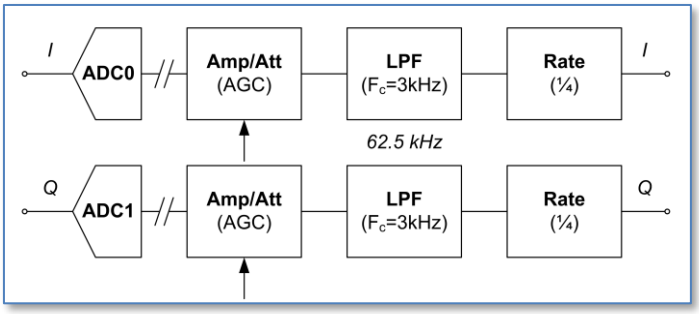
Signal processing (dsp.c)

RX stream



The RX stream can be functionally segmented into a part that does the sampling and decimation, a part that does the actual demodulation type of choice, and a part that does the audio generation. The output gives rise to an AGC feedback, where the amplification of the input is changed in factors of 2. The AGC mechanism is only local, the RX front-end optionally includes larger scale attenuation of pre-amplification of the received signal.

Sampling



The RX stream function is called every 16usec, where the most recent I and Q samples are taken from the buffer, resulting in 62.5kHz sampling rate for each channel.

The phase error between the I- and Q-samples is normally only the duration of an ADC conversion, 2usec or worst case 4usec depending on actual process timing. This 4usec results in a phase shift which is about 2% in the audio domain (at 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. This averaging process in fact results in a much larger distortion.

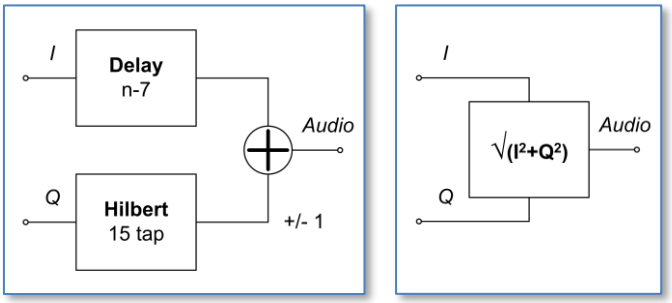
DC removal

The samples from the ADC are electrically centered on about half the reference voltage, which is half the dynamic range. After subtraction of half the ADC range there is also a DC removal process, that subtracts the low-pass filtered running average signal.

dc += (sample – dc)/128

The stream is then amplified with the AGC factor, which can be implemented as a simple bit-shift. Worst case this results in some loss of accuracy, but this is somewhat hidden in the following decimation stage. This consists of a low-pass filter and a 4x down-sampling, resulting in I and Q output streams of 15.625 ksps.

Demodulation

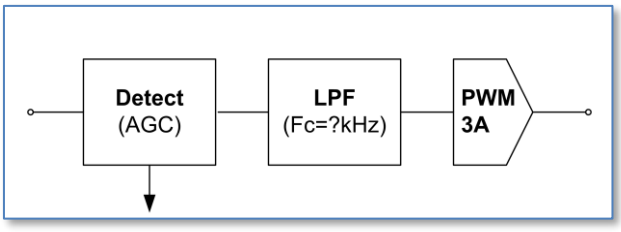


For SSB demodulation, the decimated I and Q samples are stored in a 15-sample delay line. A 15-tap Hilbert transform on the Q channel is done and the result is subtracted from or added to the n-7 sample in the I delay line to obtain the USB or LSB audio output respectively.

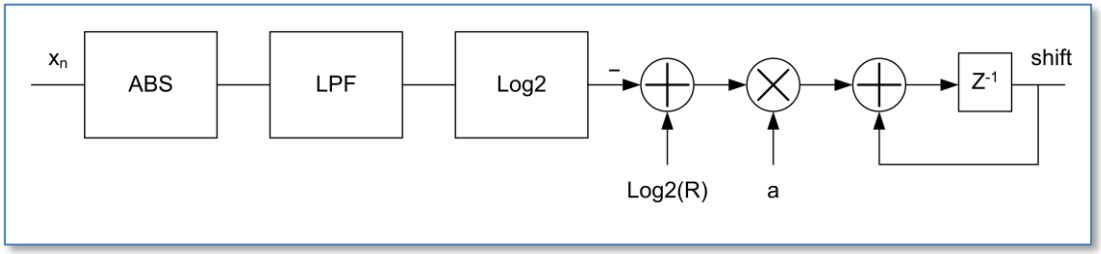
For AM demodulation the length of the I-Q vector needs to be calculated, and no further transform is required.

The resulting 15.625 ksps audio sample stream is passed to the Audio generation.

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.



The process determines signal level by taking its absolute value and then leading it through an exponential averaging (low-pass) filter:

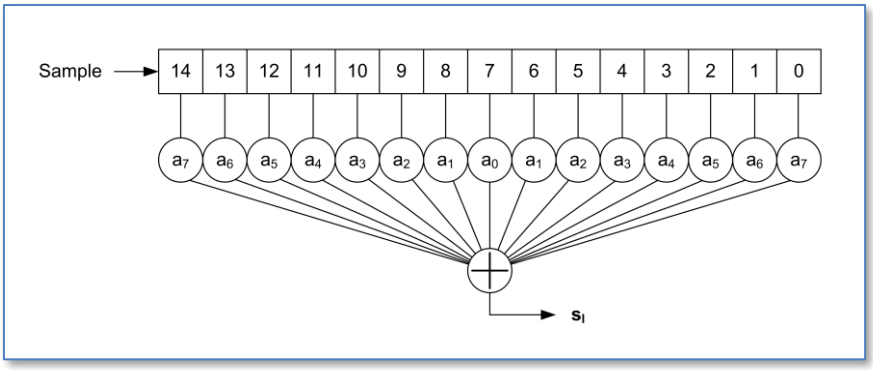
$y_n = (1-q)\cdot y_{n-1} + q\cdot|x_n|$,

where *q* can be chosen 1/8 or 1/16 depending on the required amount of filtering.

The averaged value is logarithmically compared to the desired level, which should be maximum half the DAC range. Note that the full DAC range is set to 256 to obtain high enough PWM frequency for proper analogue filtering (125MHz/256) so max value would be 127. In fact, we are steering on the most significant bit of R and X_n values. For R this is by definition bit 6 (64=0x40), from which the log2 of y_n must be subtracted. A positive value will increment a left-shift value, a negative value will decrement the left-shift value while 0 will not change the shift value. The dynamic range and accuracy are not brilliant, but the mechanism is simple and should work with five 6dB steps.

Attack / decay times can be implemented by an integrator (accumulator) with different values of a, where the time is roughly equivalent to the sampling time divided by a. So, assuming a sampling frequency of 15.625kHz and a=0.01 (1/128) the attack/decay time is 6.4msec, which would be an okay attack time for the fast AGC setting. A value for slow AGC setting could for example be 0.001 (1/1024). Every time the attack/decay time threshold is passed, the gain is incremented or decremented and the accumulator reset.

Low Pass Filters



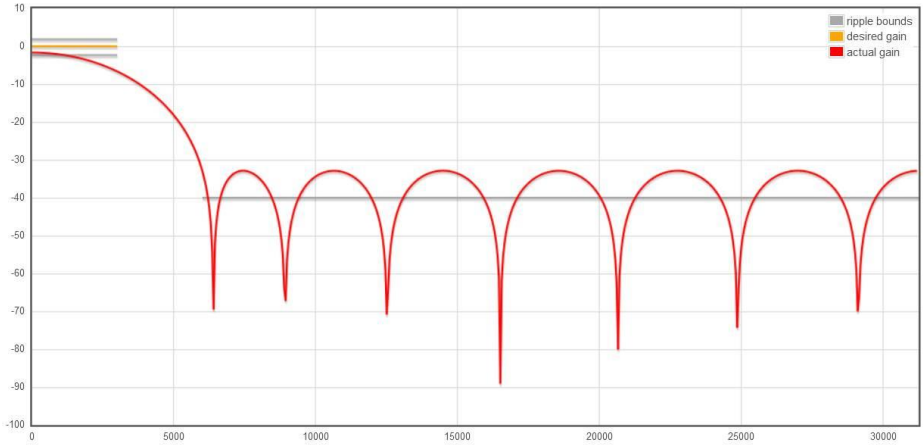
Several 15-tap low pass filters have been created, with a corner frequency of F_c=3kHz. The stop-band depends on the actual sample rate the filter is designed for. It usually starts around 5kHz, with a level of -40dB or better. The code contains filters for 62.5 kHz, 31.25 kHz and 15.625 kHz sample rates.

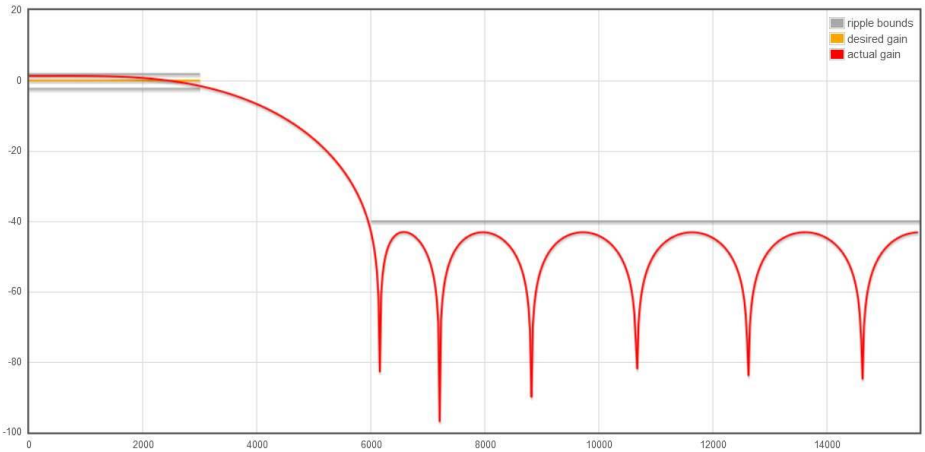
These low pass filters are simple symmetric FIR filters, that represent the impulse response of the desired low pass behavior. They consist of 15 signed integer arrays. Hence, per sample 15 multiplications and additions need to be done, but the RP2040 has a single cycle 32bit MPY instruction, so that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

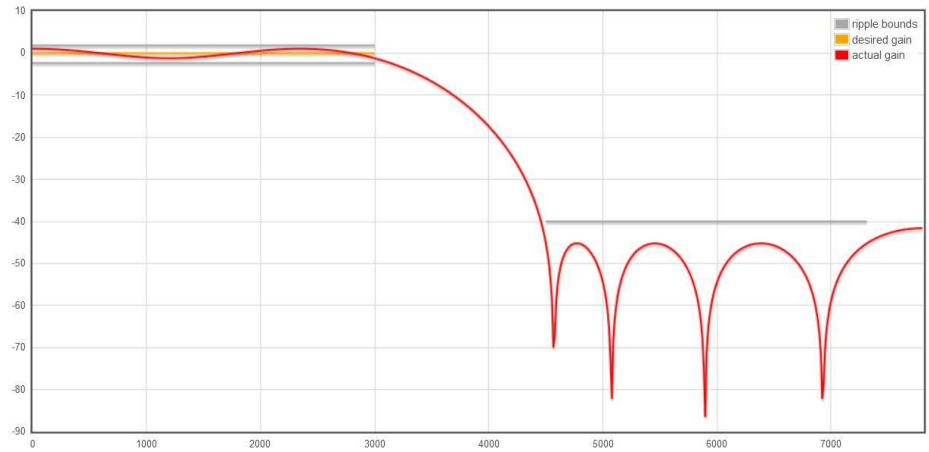
For coefficients see for example Iowa Hills DSP tools or the T-Filter on-line calculator:

- 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:

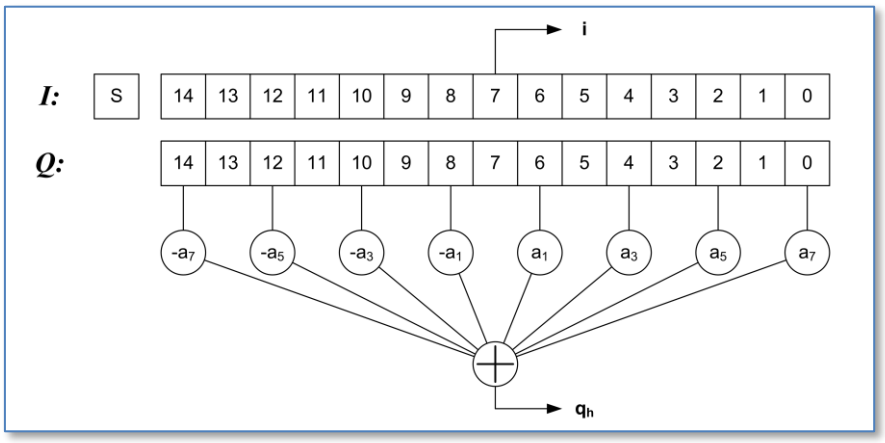






Note that for the 3kHz low pass filter, a 15 tap FIR algorithm works best at lower sample rates. For high sample rates it would be better to use more taps.

Hilbert transform

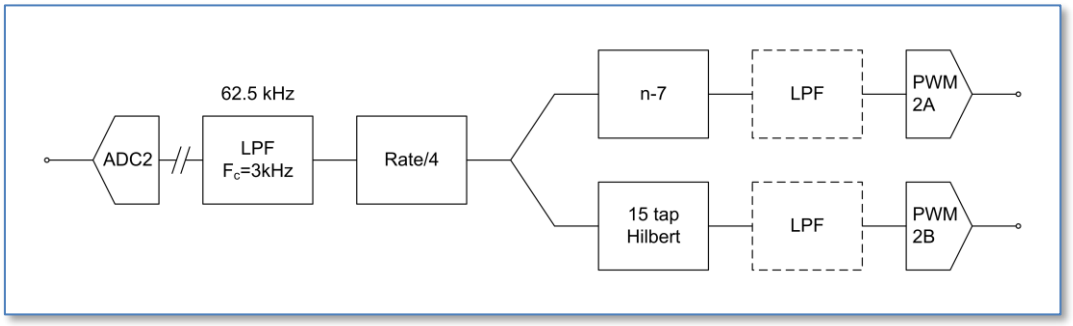


The Q delay line array has a length of 15, to enable a 15-tap classic Hilbert transform. The even samples have a zero coefficient so, as in the above figure, only 8 of the samples are used in the calculation. Due to symmetry of this classic Hilbert transform only 4 multiplications have to be performed. 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 Iowa Hills tools to obtain a set of coefficients.

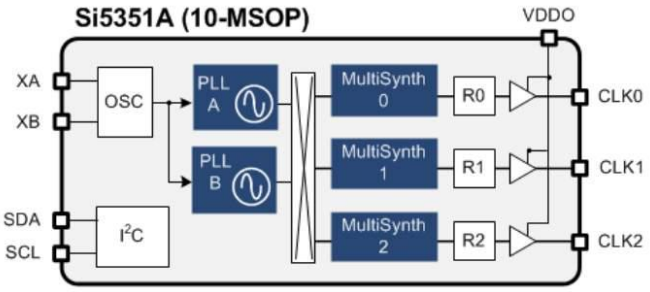
Note that the bandwidth of the classic Hilbert transform is half the sampling frequency. The response at the edges drops off, so to get a response that extends far enough towards 0 either the sampling rate must be lowered or the number of taps must be increased. This is one of the reasons for inserting the decimation stage.

TX stream



The TX stream is the inverse of the RX stream: the same components can be found here as in the RX stream.

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 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.

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

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. When in MENU state, only a subfunction can be selected. The choice is made by either the encoder or by the left-right buttons. The Enter key causes to enter the subfunction, and also the related state.

States:

- Menu
- Tune (change VFO frequency)
- Mode (USB, LSB, AM, CW)
- AGC (Fast, Slow, Off)
- Pre amp (Amplifier, Attenuator, off)
- XMT

Combining States and Events we can create a matrix, but many events have similar actions so that may not be the best implementation. So, per state:

Menu

Left, Right Inc and Dec cycle through the subfunctions (i.e. other states). Enter selects and changes state, other events do nothing.

Tune

Left and Right select digit, Inc and Dec change the value of the digit and hence the set frequency. Enter accepts the new value while the Tune function is not left. Cancel leaves the function without accepting the new value.

Mode, AGC, Pre

Left, Right, Inc and Dec change the selection. Enter accepts the new selection and exits the function. Escape leaves the function with no change.

XMT

The PTT active event forces this state, whatever the previous state was. The state is left when the PTT is released, to avoid additional complexity the next state is always Tune, since this is probably most frequently used.

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.

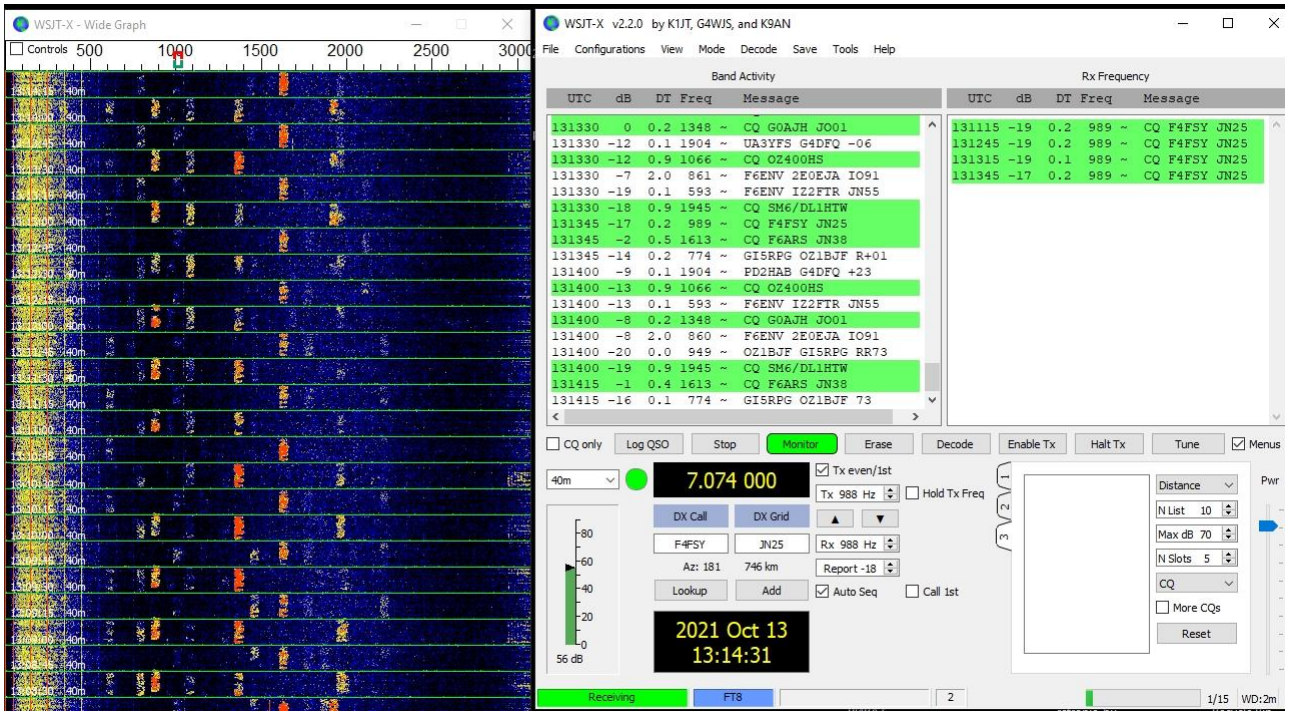
To enable stdio, a call has to be made to `stdio_init_all()`. This is actually done inside the monitor initialization routine, so best to call this early in the start-up sequence.

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.

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.

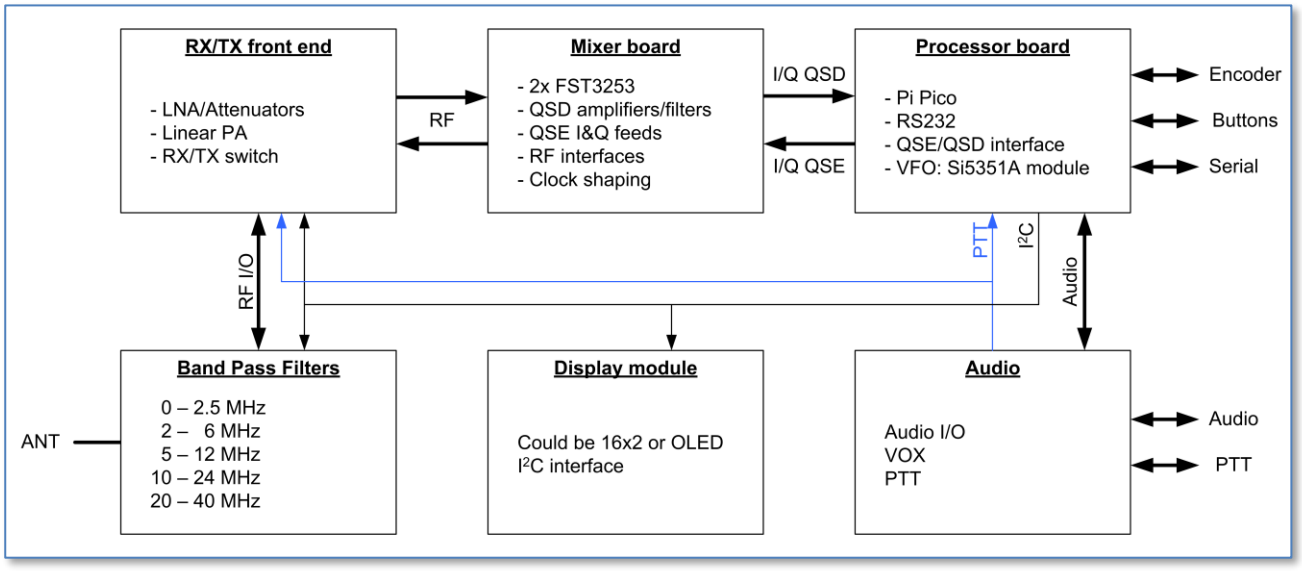


As the screendump shows, the current version works nicely on the 40m FT8 channel. The AGC now also works, albeit not yet controllable from the menu. It makes sure that the output signal is maintained between 1.6V_{pp} and 3.3V_{pp}.

Proto implementation

The uSDR prototype will be based on off the shelf modules and will end up in a Teco 1500 enclosure. It may serve as a test environment for the realization of a more integrated implementation on the longer term. However, the modularity of the prototype is quite good for experimentation, since it allows to swap out certain functions of the system that do not function appropriately.

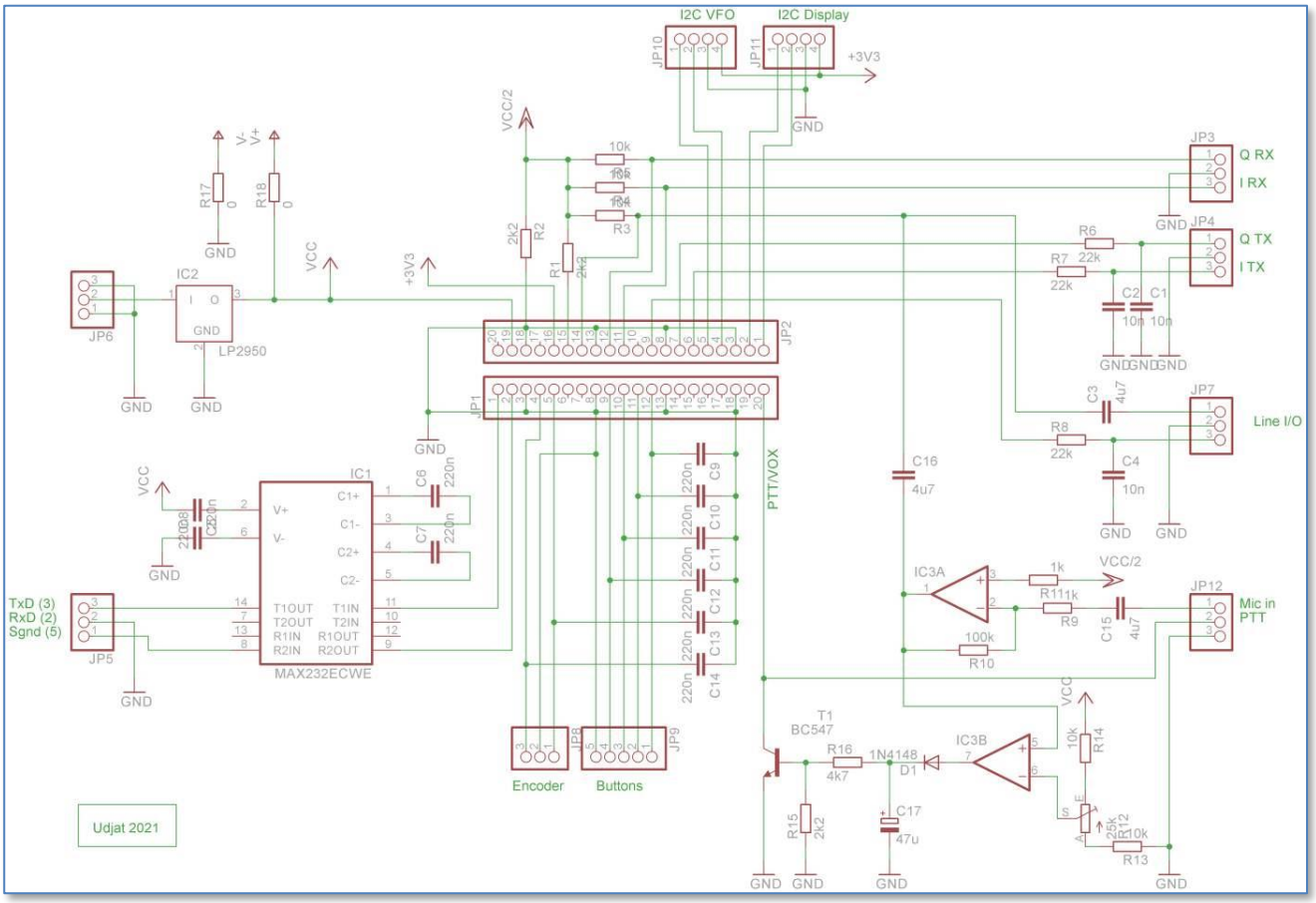
The modular architecture is conceived as follows:



In summary, the modules are:

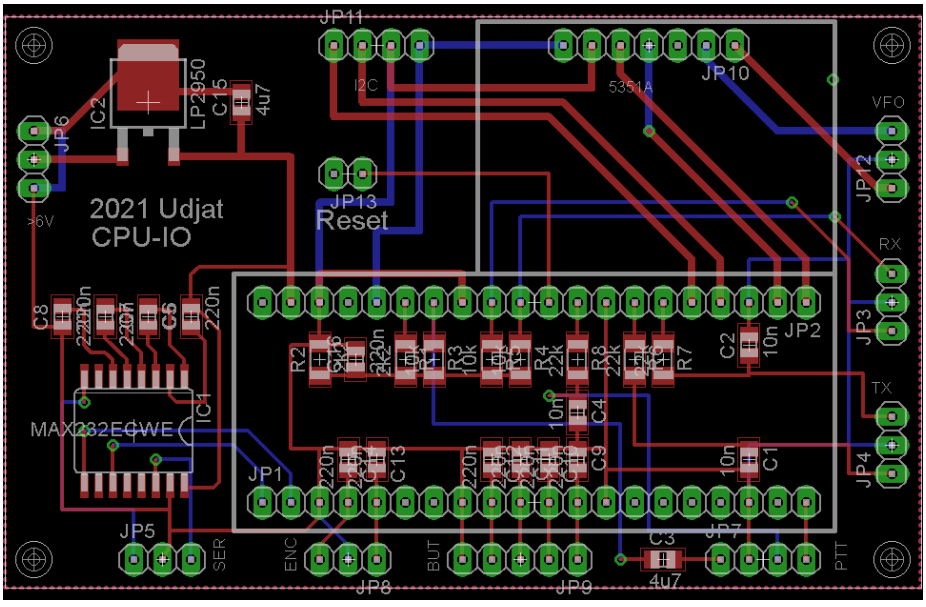
- Processor board: This is simply a carrier for the Pi Pico as well as the VFO module. The interfacing comprises the RS232 levelling, switch debouncing and PWM/DAC filtering
- Audio board: This contains all analogue audio handling, VOX and external PTT interfaces.
- Display module: The display module can be anything suitable with an I^2C 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 end: This board contains the TX PA as well as the RX LNA and attenuators. The switching between RX and TX paths is done by PTT controlled relays.
- Filter board: This contains a set of 5 switched bandpass filters, that go in between RF front-end and antenna.

Processor board

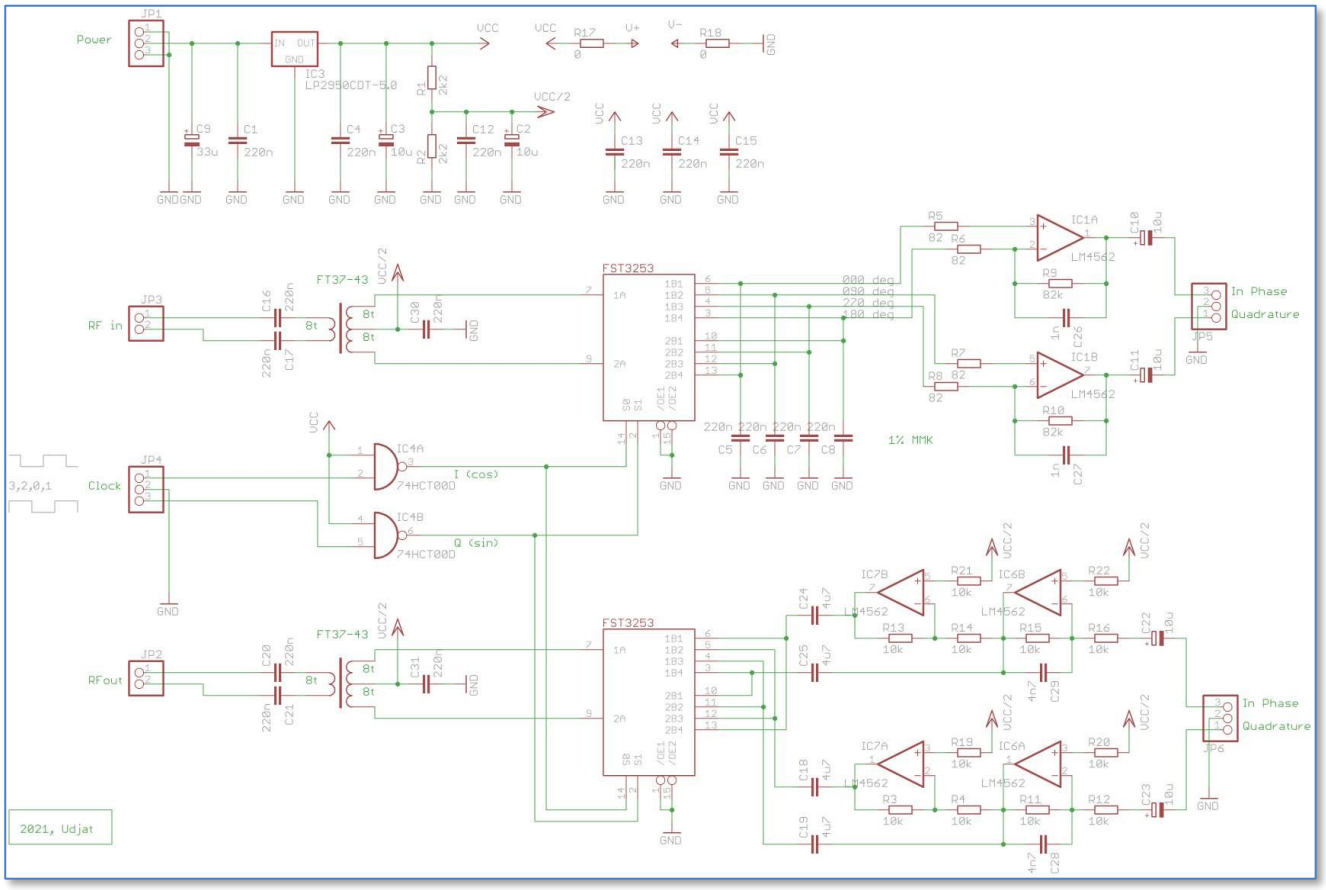


The processor board hosts the Pico module and all user I/O. There is also a serial interface that receives a monitor with a command line interface. The PTT is generated either from a hard signal from a microphone, or by a level detection circuit in case of e.g. computer generated FT-8 encoding.

Board of appr 50x81mm:

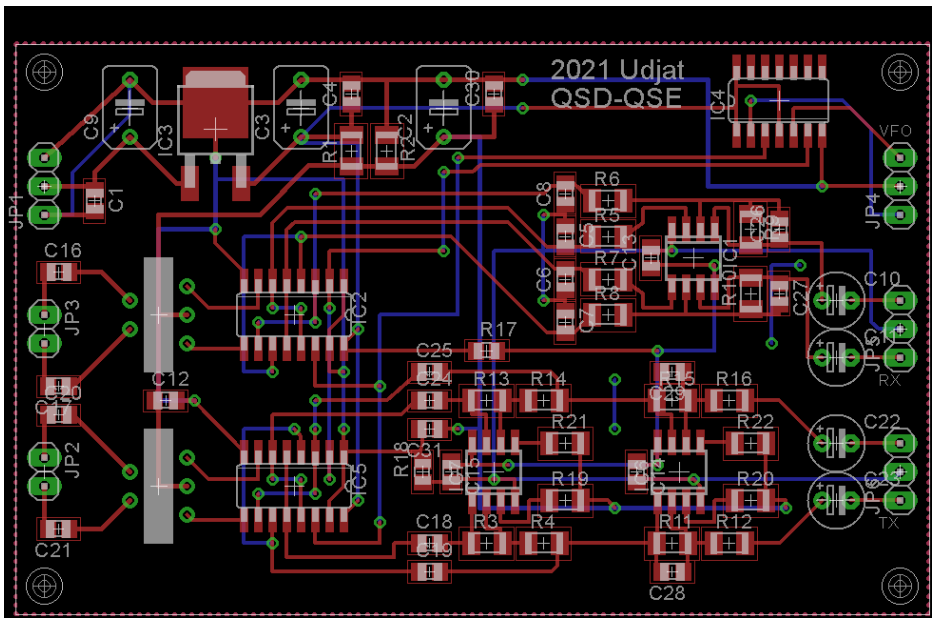


Mixer board

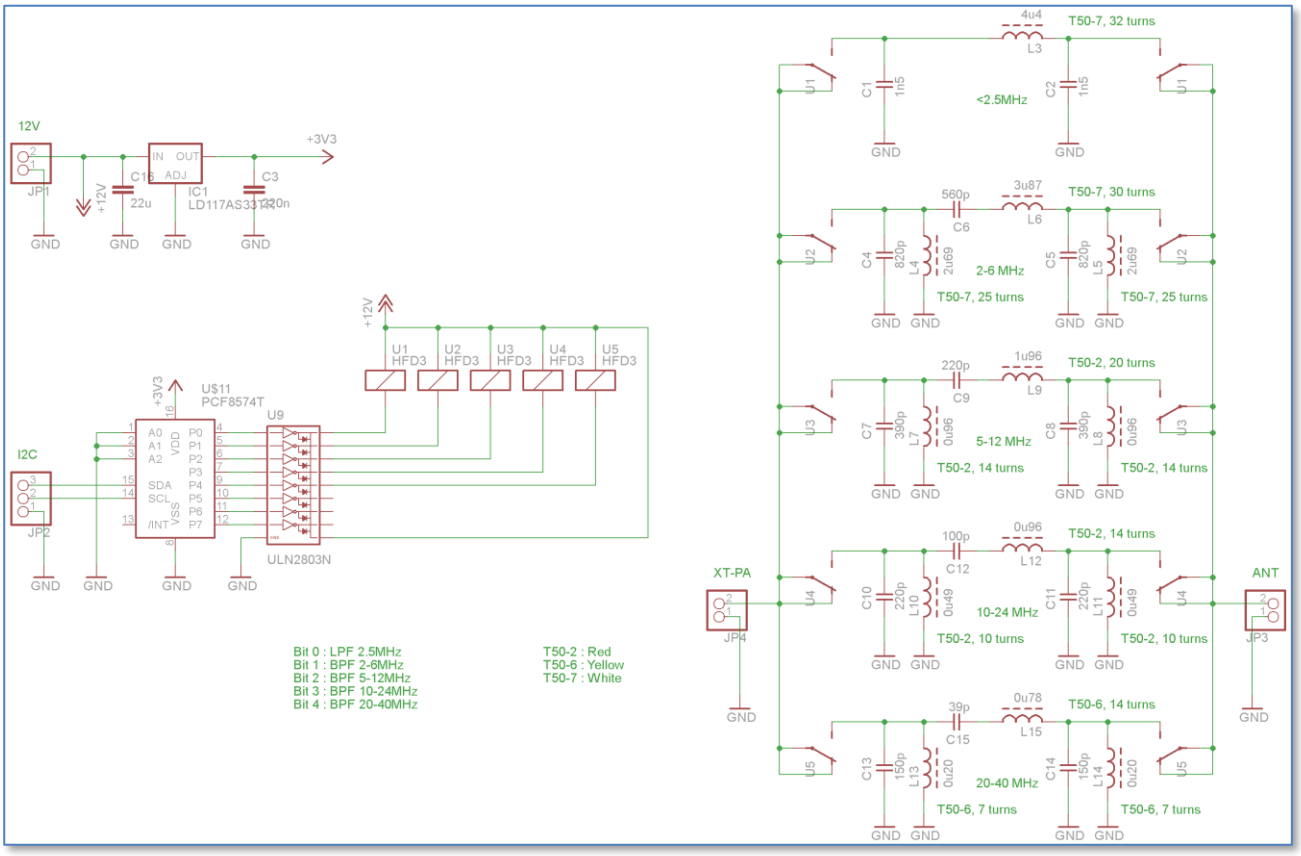


The mixer board hosts the two mixers, QSD for RX branch and a QSE for the TX branch. The switches are clocked from the Si5351 board, through two NAND gates that take care of level shifting and some signal cleanup.

Board of appr 50x81mm, clicks behind processor board:

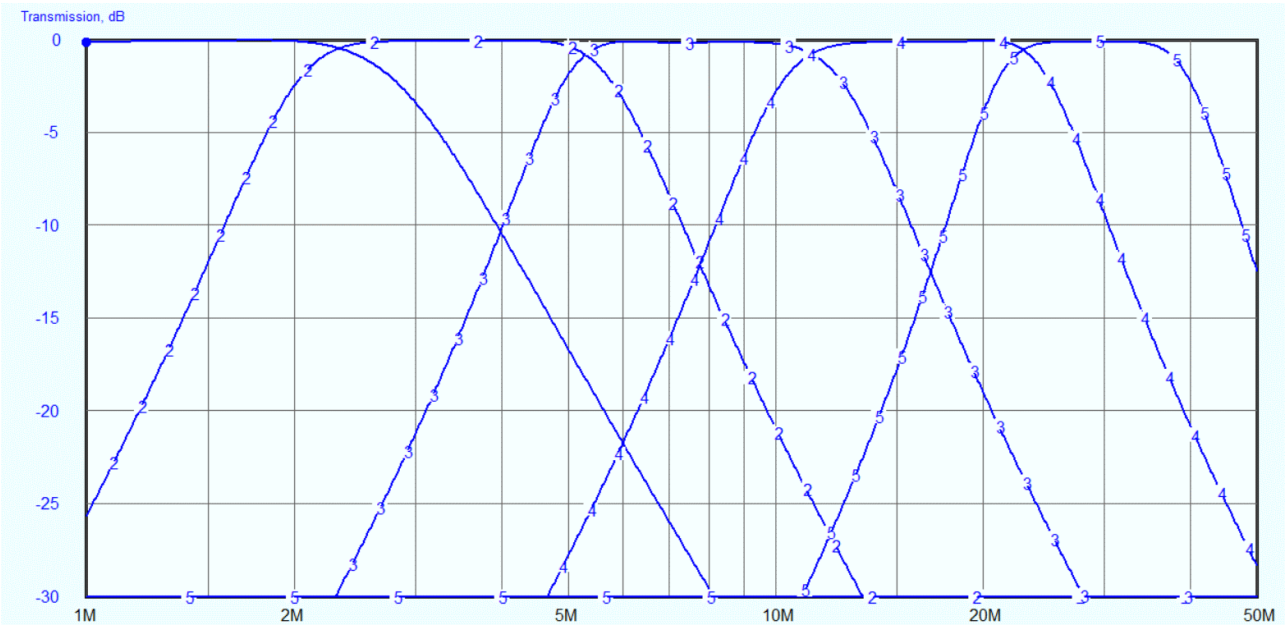


BPF board

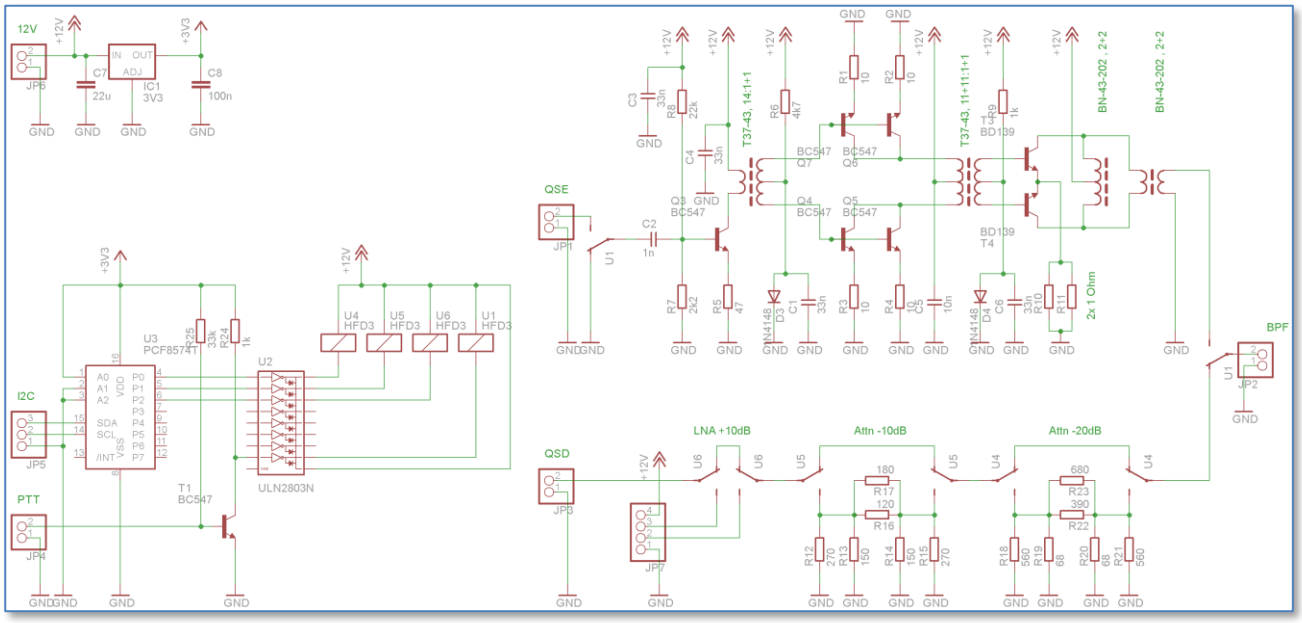


The 5 band filters and the attenuators/LNA are selected through relays, controlled from the Pico via the I^2C bus.

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



RX/TX board



The RX/TX board is still highly under development. The TX path contains a wideband linear driver as proposed by Harry Lytthal (SM0VPO) that outputs up to 500mW. The end-stage, also class AB push-pull configuration, should be good for about 5 Watts. The symmetry of this circuit should better suppress the even harmonics.

The RX path contains the selectable attenuators and low noise amplifier.

RX/TX switching is controlled directly from the PTT signal, connections to QSD and QSE are separate.

Work in progress, more details will follow, PE1ATM.

EOF