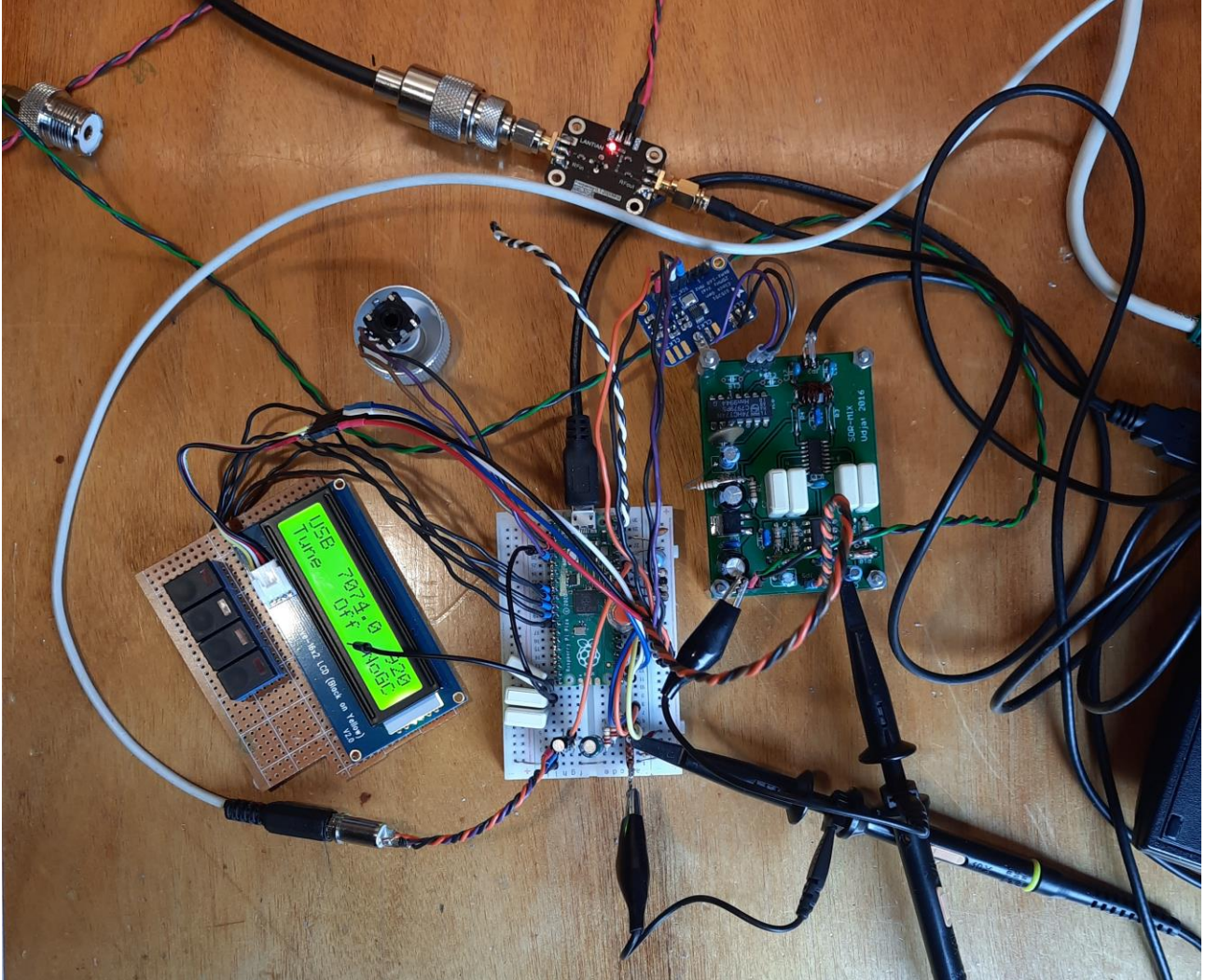# Micro SDR on a Pi-Pico

Arjan te Marvelde, May 2021

Since 2020, Raspberry offers a new board based on their own development processor RP2040. This contains a dual core, 125MHz Cortex controller with plenty Flash and RAM. The processor has many highly configurable IO pins, which make application of the module a breeze. The C-SDK is still somewhat immature, but at least provides a quick start in actually setting this Pi-Pico to use.

This article describes a test implementation of a small SDR, inspired by Hans Summers' QCX and everything that followed after that.

The image above shows the test setup at some stage where the RX part is working. The project can be found on and downloaded from GitHUB.

Raspberry Pi Pico (RP2040)

The Pi-Pico is the basis for the implementation, so here is the pinout and the way it is used within the project.


Pi Pico pin usage for uSDR project
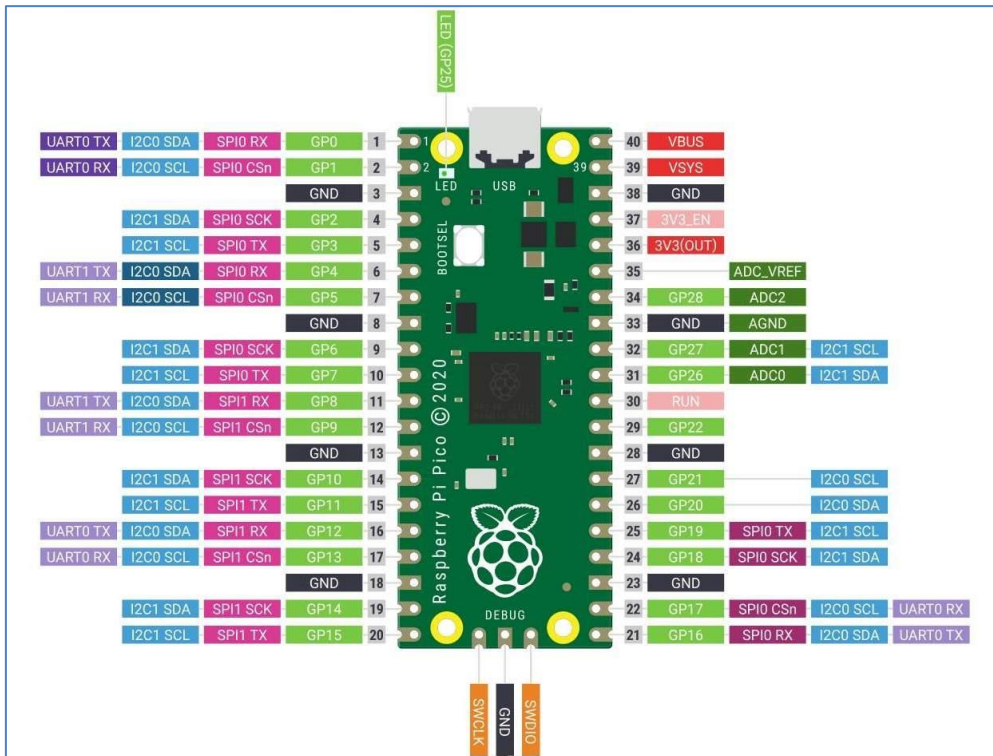
UART0 Tx	GP0	Vbus		Not used
UART0 Rx	GP1	Vsys		5V power input
	GND	GND		
Encoder A input	GP2	3V3 en		Not used
Encoder B input	GP3	3V3 out		3V3 to peripherals
Not used	GP4	Va ref		ADC 3V3 reference output
Not used	GP5	GP28		ADC2: Audio input
	GND	GND		
Aux button 1 input	GP6	GP27		ADC1: QSD Q-channel 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		
BPF select output	GP10	GP21		PWM 2B: QSE Q-channel output
BPF select output	GP11	GP20		PWM 2A: QSE: I-channel output
BPF select output	GP12	GP19		I2C1 SCL: Si5351A
BPF select output	GP13	GP18		I2C1 SDA: Si5351A
	GND	GND		
Squelch output	GP14	GP17		I2C0 SCL: LCD screen
PTT input	GP15	GP16		I2C0 SDA: LCD screen

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, several auxiliary buttons, a serial port for monitoring and an LCD on the other I^2C interface.
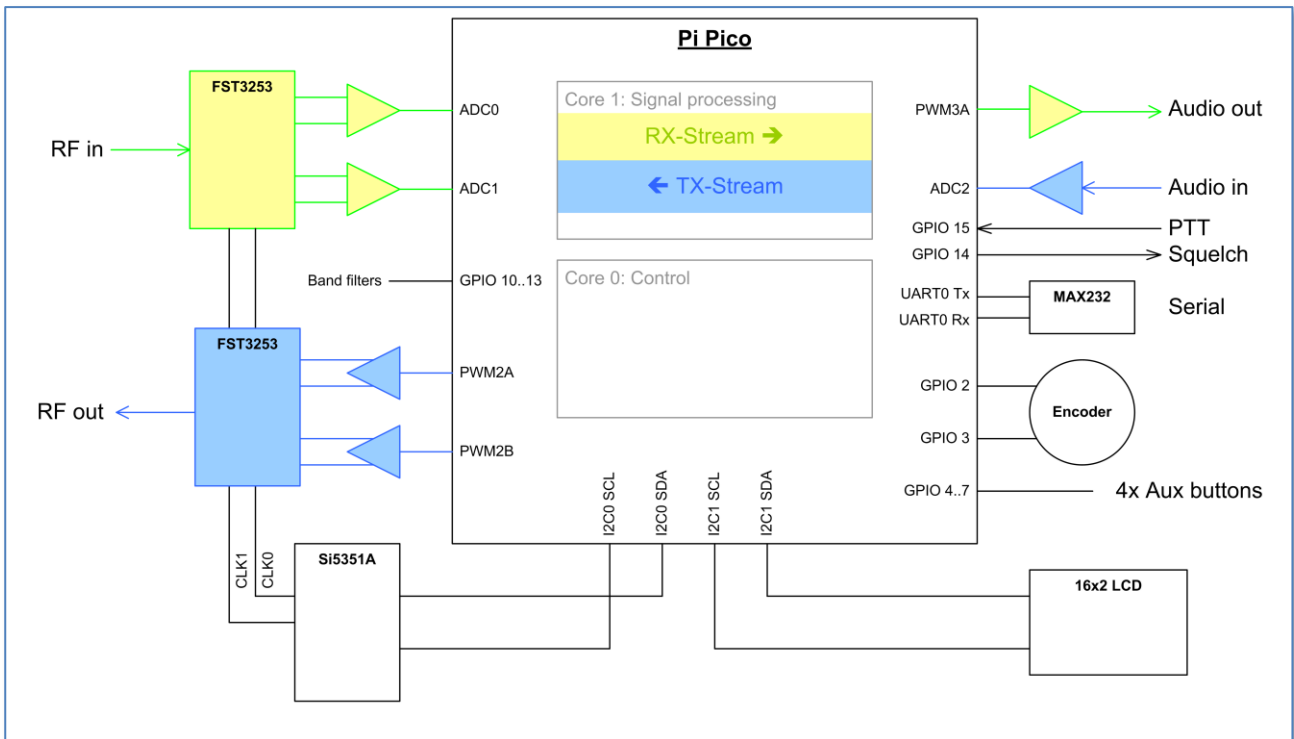
Code layout

The processor has two cores that can process mostly independently. The idea is to let *core1* do all the signal processing, while *core0* (the default) does all the control stuff.

The signal processing on *core1* is basically split up in two streams, and RX and a TX stream. The three ADC inputs are sampled continuously in Round-Robin fashion, and the interrupt handler merely copies the conversion results into buffers. This way, ADC samples are taken every 6us for every channel. 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. The TX stream does the reverse, taking the samples from the audio input, process them and send 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 only pushes a mode word (being TX or RX) through the inter-core fifo, which subsequently sets off the right process in *core1* (so either TX or RX processing).

The Pico has two I^2C buses, so we can make it easy and connect one device to each. There are four GPIOs reserved for auxiliary buttons, and four more for switching band pass filters. 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
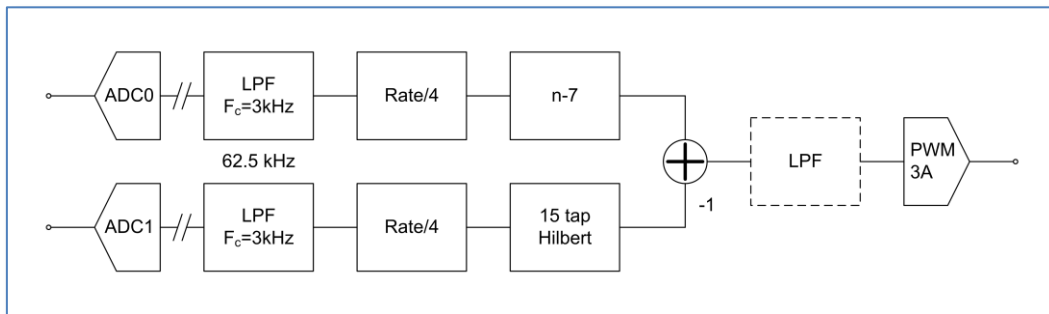
The RX stream function is called every 16usec, where the most recent I and Q samples are taken from the buffer, resulting in 62.5kHz sampling rate for each channel.

The phase offset of the I-samples is normally only the duration of an ADC conversion, 2usec or worst case 4usec depending on 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.

The raw I and Q samples are stored in a delay line, ready for low pass filter and decimation to 15.625 kHz rate. For USB mode a 15-tap Hilbert transform on the Q channel is done and the result is subtracted from the n-7 sample in the I delay line to obtain the audio output. For AM mode the length of the I-Q vector needs to be calculated, and no transform is required.

The resulting samples are passed to the Audio output DAC. Optionally another LPF can be inserted.

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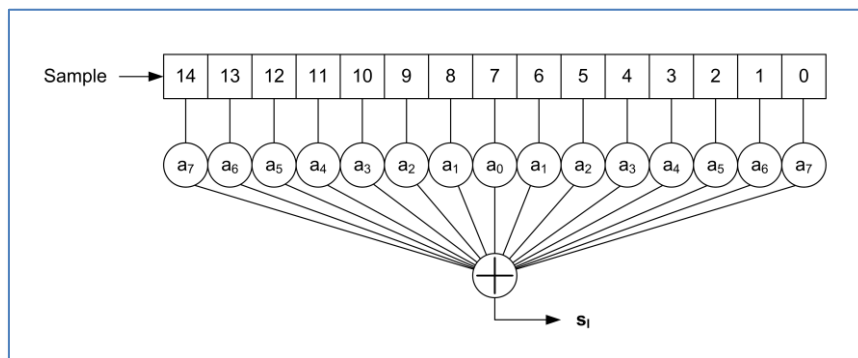
Several 15-tap low pass filters have been created, with a corner frequency of F_c=3kHz. The stop-band depends on the actual sample rate the filter is designed for. It usually starts around 5kHz, with a level of -40dB or better. There are filters for 62.5 kHz, 31.25 kHz and 15.625 kHz sample rates.

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. Per decimated sample 15 multiplications must be done, but the RP2040 has a single cycle 32bit MPY instruction, so that should be pretty fast.

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

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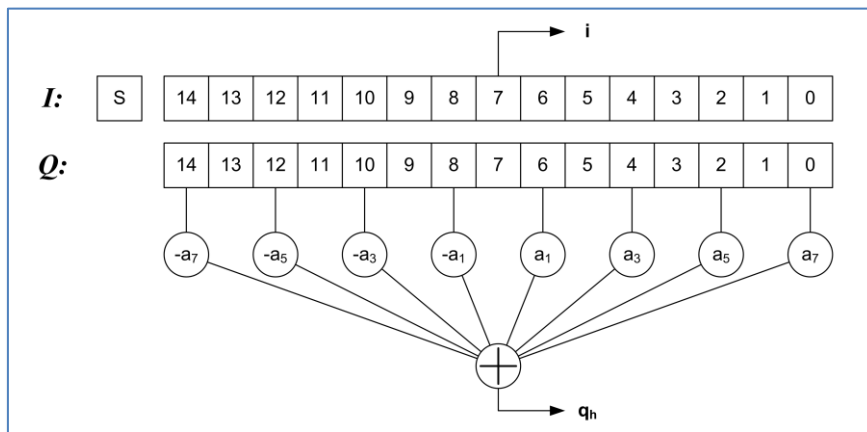
The Q delay line array has a length of 15, to enable a 15-tap classic Hilbert transform. The even samples have a zero coefficient so, as in the above figure, only 8 of the samples are used in the calculation. Due to symmetry of this classic Hilbert transform only 4 multiplications have to be performed.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

Demodulation

The I_{n-7} and Q_h (or I and Q) samples are used to obtain the desired audio signal. For SSB demodulation (USB) the I_{n-7} and Q_h samples are simply subtracted.

AGC and RSSI

The automatic gain control (AGC) and received signal strength indication (RSSI) should be added still
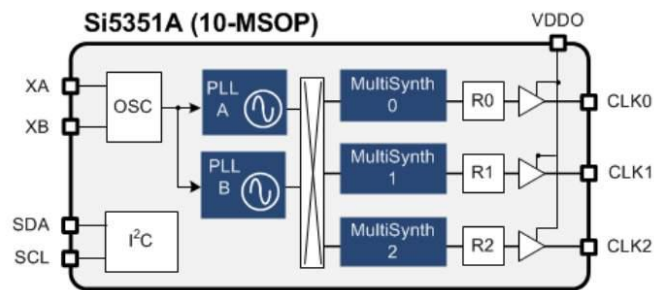
It may be better to implement this outside the Pico, since the circuit best ensures the ADC input is up to full range.
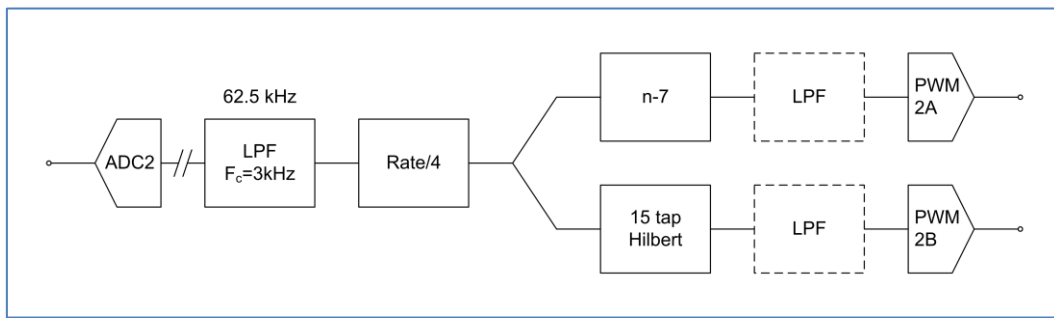
The TX stream is the inverse of the RX stream: the same components can be found here as in the RX stream.

Quadrature VFO (si5351.c)

The Si5351A is a triple clock generator, that can be controlled through I^{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).

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:

- R_i=128 for F_{out} <1 MHz
- R_i=32 for F_{out} 1-6 MHz
- R_i=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 difference of VFO i to (p = {0, 1, 2, 3} x 90deg)

`SI_SETPHASE` obviously only works for VFO 0.

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.

Test circuit

For testing purposes and SW development the Pi Pico was plugged into a breadboard and peripherals are added as needed. A few points of attention:

- The ADC inputs need to be biased, I just added 2x10k connected to GND and +3V3. The latter should ideally be a proper reference voltage.
- The PWM/DAC outputs need to be filtered through a 22k/1nF low pass network, to suppress the 500kHz PWM frequency. Then coupling to external circuits should be done with a capacitor of about 10uF.
- I made a reset button between pin30 and pin28, to make reset and transition to loading mode easier.
- Some peripherals can be 5V powered, take care of any pull-ups on e.g. I^{2}C if you do. The 5V will destroy the I^{2}C interface on the Pico!
- Si5351 is set to generate 180deg double frequency clock, due to the used legacy mixer implementation.

The RX mixer I still had from a previous project is based on a FST3253. The RF input is a transformer with 3x 8 turns. The sampling capacitors are 100nF, and the MCP6231 OpAmps have 30dB gain (82Ohm input and 82kOhm feedback).

The RF input is amplified with a wideband MMIC LNA. The gain can be controlled with the supply voltage, and should be chosen so as to generate a max 3V3 swing on the mixer outputs.
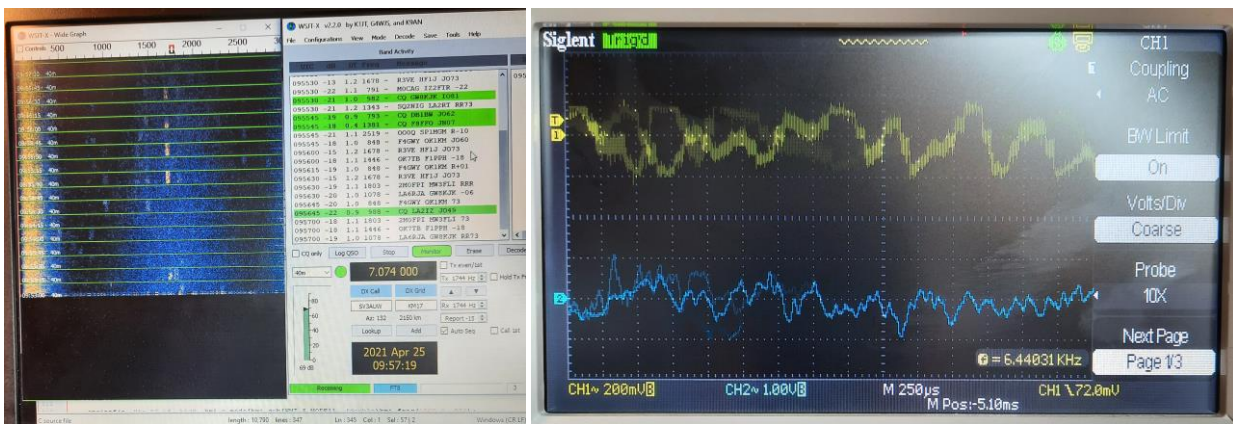
Left picture is a WSJT-X run, on a bad propagation time of the day. In the evening the band is filled about 50%.

Right picture is a snapshot of the I or Q signal from the mixer (bottom) and the audio output to the PC (top).

The audio output looks a bit noisy, probably due to the 500kHz PWM still shining through, this might be solved by a better (active) filter.

Proto implementation

A 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:
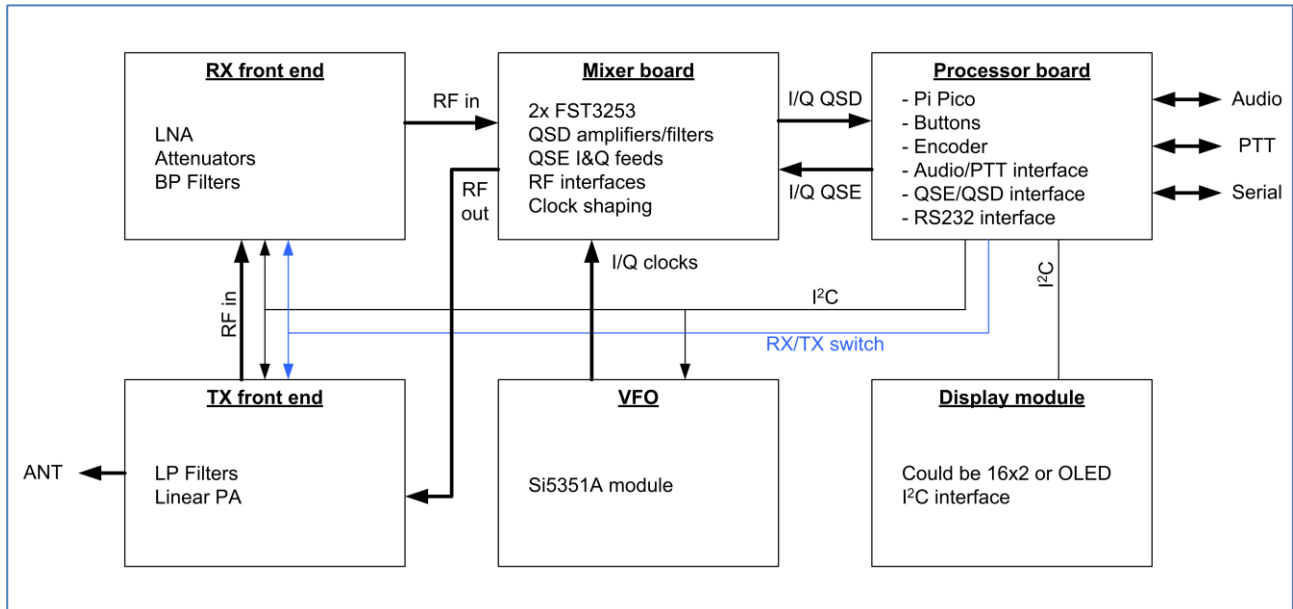
In summary the modules are:

- Processor board: This is simply a carrier for the Pi Pico, and all interfacing electronics. This interfacing comprises the RS232 levelling, audio amplifiers, switch debouncing and PWM/DAC filtering
- Display module: The display module can be anything suitable with an I^2C interface. The SW is made for a regular 16x2 alphanumeric type.
- VFO: The VFO module is implemented with an Adafruit Si5351A module, which is controlled over I^2C.
- Mixer board: This is a design based on two FST3253 switches. Clock shaping is done with a 7400 ACT or HCT, and analogue signal handling with three LM4562.
- RX front end: This module amplifies or attenuates the RF input, and passes it through appropriate Band Pass Filters. All switchable parts can be controlled from the Pico over the I^2C bus, through a PCF8574 expander. Possibly the RSSI measurement and AGC can also be implemented through I^2C controlled ADC/DAC.
- TX front end: This module is the Power Amplifier for the TX signal, and also contains switchable filters for cleaning up the output. The Antenna relay can be situated here as well.

Processor board
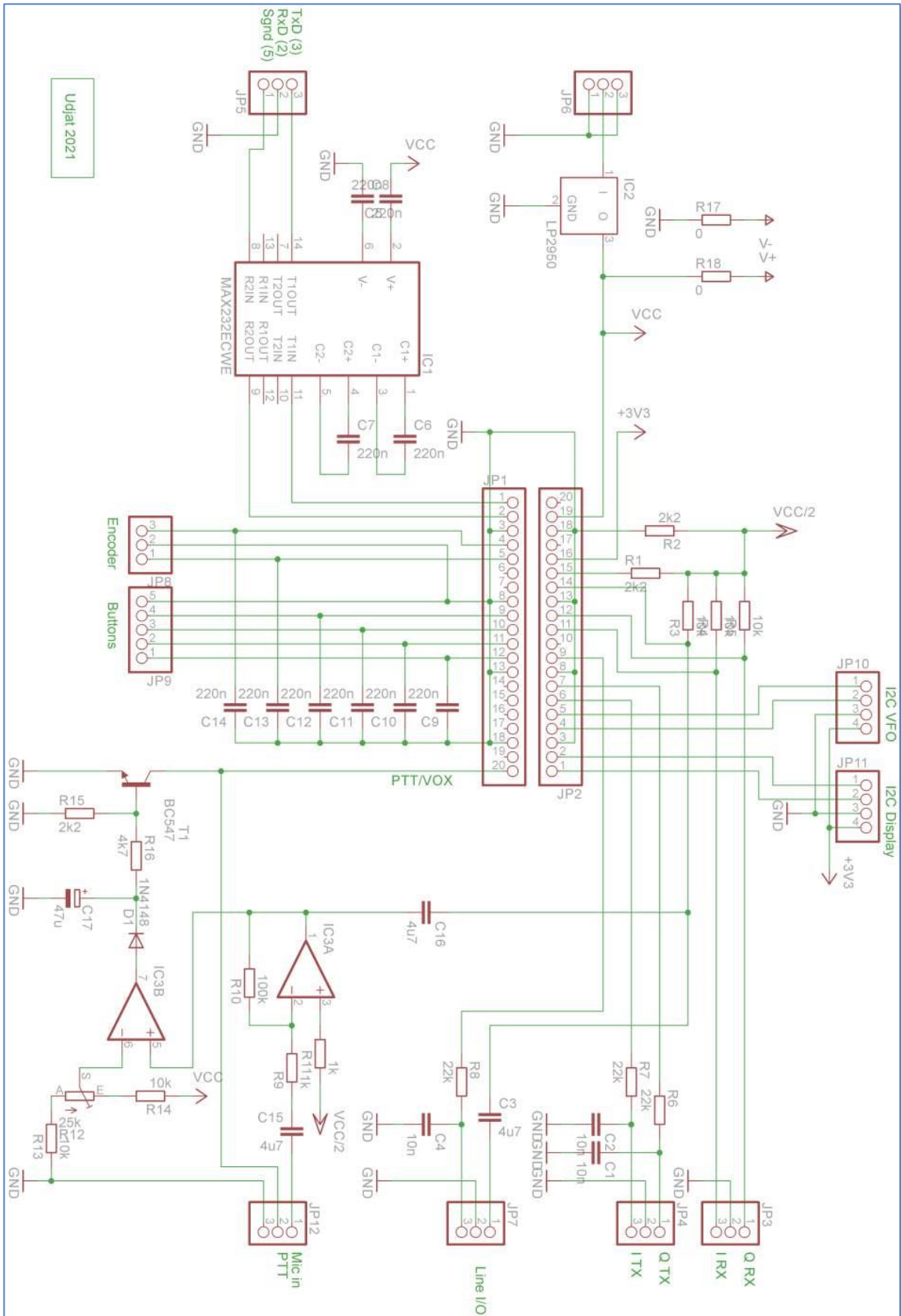
Mixer board
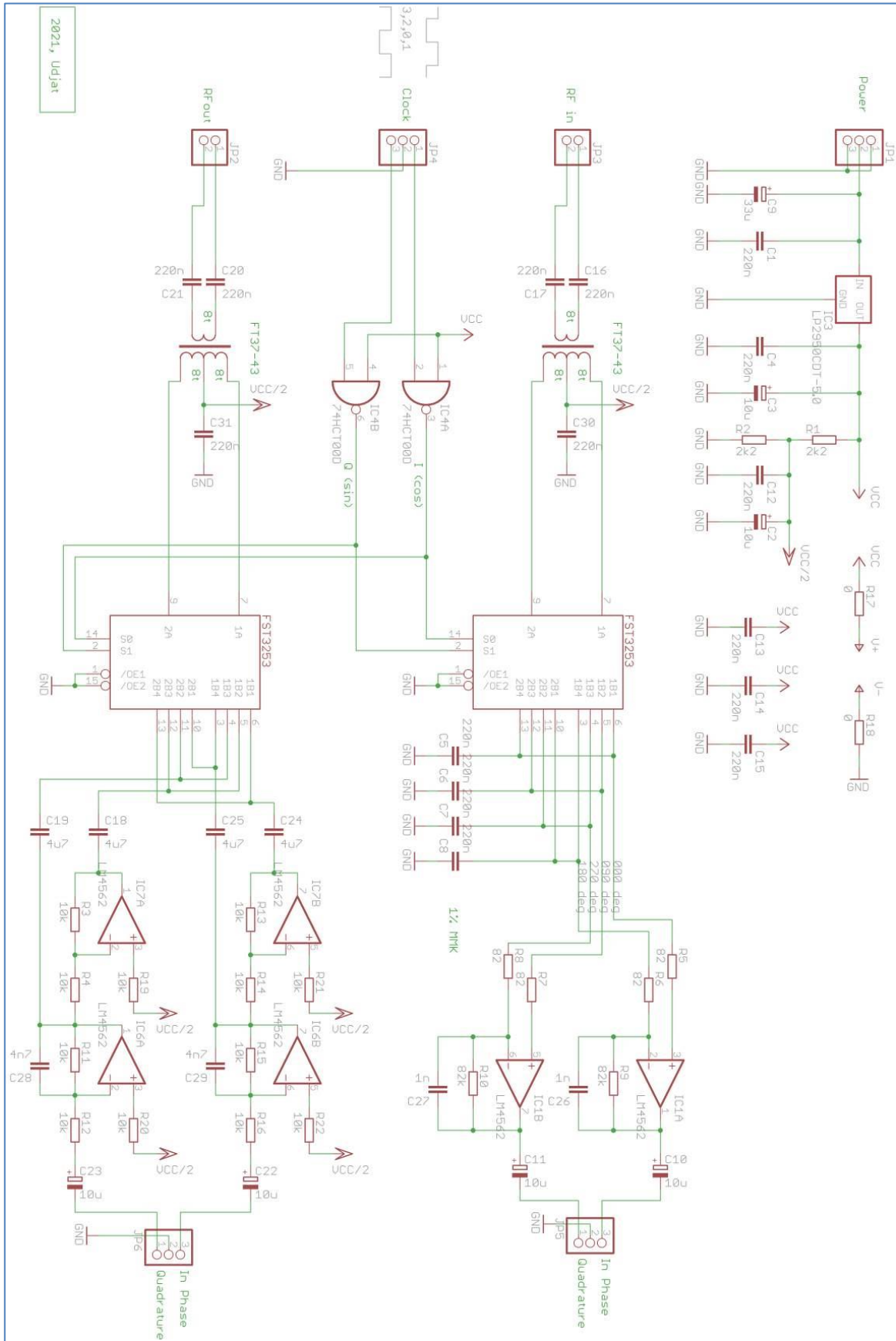
More details will follow.

PE1ATM

EOF