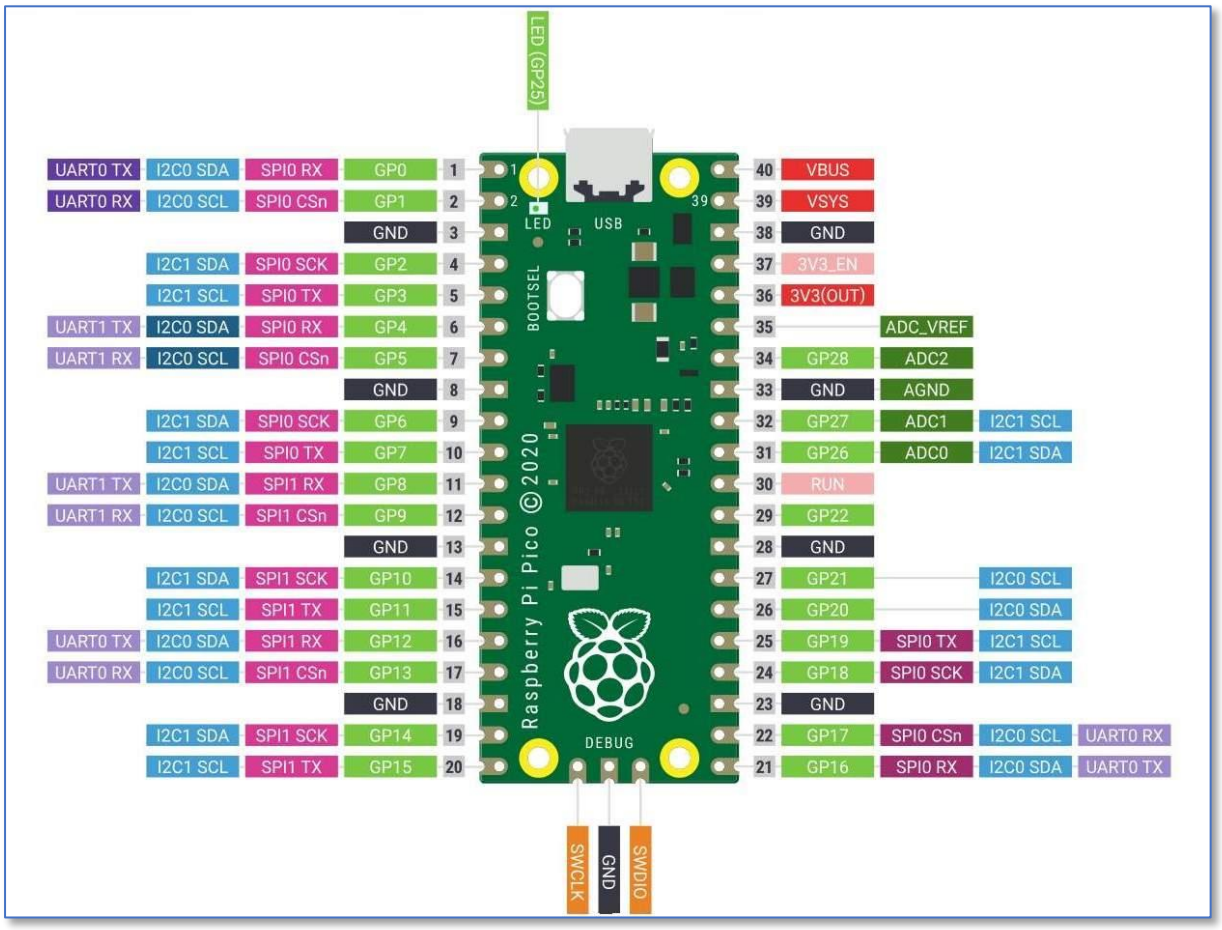
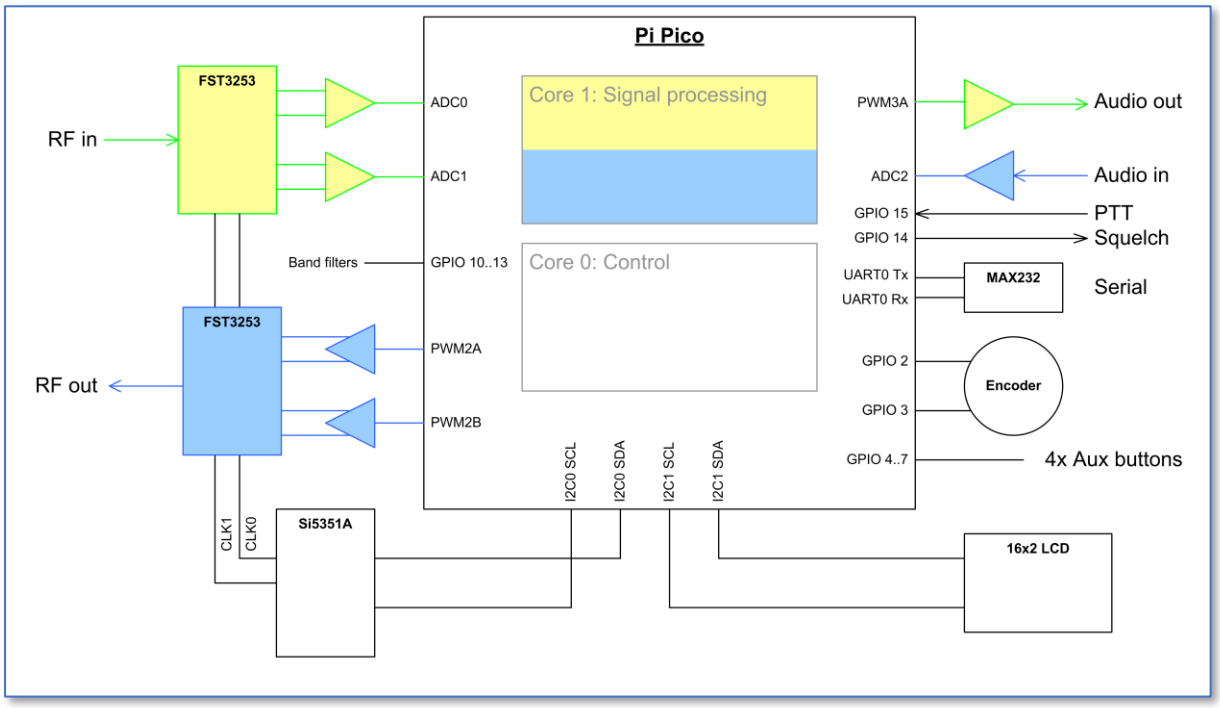
Raspberry Pi Pico (RP2040)



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
Aux button 1 input	GP4	Va ref	ADC 3V3 reference output
Aux button 2 input	GP5	GP28	ADC2: Audio input
	GND	GND	
Aux button 3 input	GP6	GP27	ADC1: QSD Q-channel input
Aux button 4 input	GP7	GP26	ADC0: QSD I-channel input
Not used	GP8	RUN	Pushbutton to ground for reset
Not used	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



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 basically is split up in two streams, and RX and a TX stream. The RX stream samples the I and Q channels input from the Quadrature Sampling Detector (QSD), processes this depending on the mode, and outputs samples through a PWM-based DAC. The TX stream does the reverse, sending I and Q signals to the Quadrature Sampling Exciter (QSE).

Both streams run on an 16usec basis (i.e. 62.5 kHz) which is controller 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.

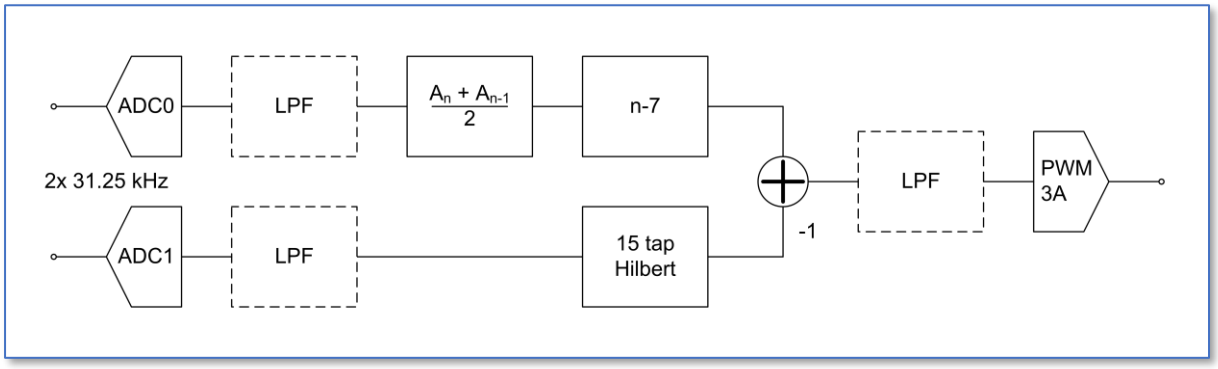
The Pico has two I2C buses, so we can make it easy and connect one device to each. There are four GPIOs reserved for auxiliary buttons, and four more for switching band pass filters. These could be used as direct controls, or with a binary decoder to increase the number of control lines.

Overview of 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.
monitor.c	The user command shell running on the stdio UART.

Signal processing (dsp.c)

RX stream

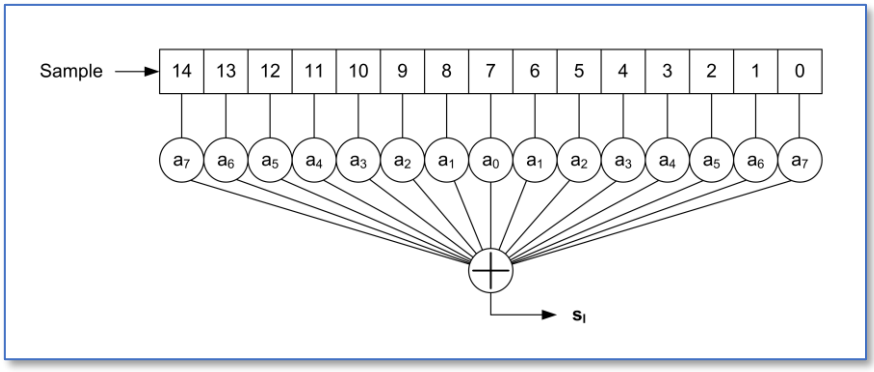


The Rx stream function is called every 16usec, but the I and Q channels are intermittently sampled resulting in 31.25kHz sampling for each channel.

The phase offset of the I-samples needs to be corrected, so they are matching the Q-samples. This is done by averaging the last two I-samples and storing this in the delay line. It is assumed that the samples are sufficiently filtered, otherwise a LPF can be inserted before phase correction.

The Q-samples are stored in the delay line directly, and a 15 tap Hilbert transform is calculated at the end of every I sample stage. It is subtracted from the n-7 sample in the I delay line, and passed to the Audio output DAC. Optionally a LPF can be inserted here as well.

Low Pass Filter

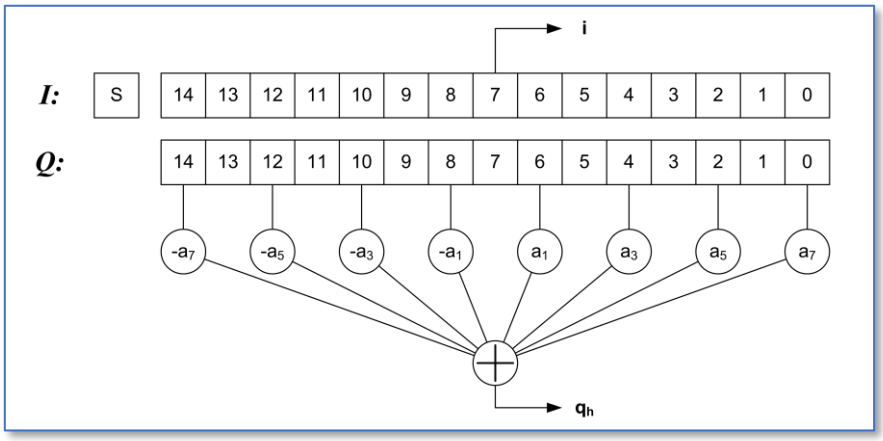


Several 15-tap low pass filters have been created, with a corner frequency of F_c=3kHz. The stop-band depends on the actual sample rate the filter is designed for. It usually starts around 5kHz, with a level of -40dB or better. There are filters for 62.5 kHz, 31.25 kHz and 15.625 kHz sampling rates.

These low pass filters are simple symmetric FIR filters, that represent the impulse response of the desired low pass behaviour. They consist of 15 signed integer arrays.

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.

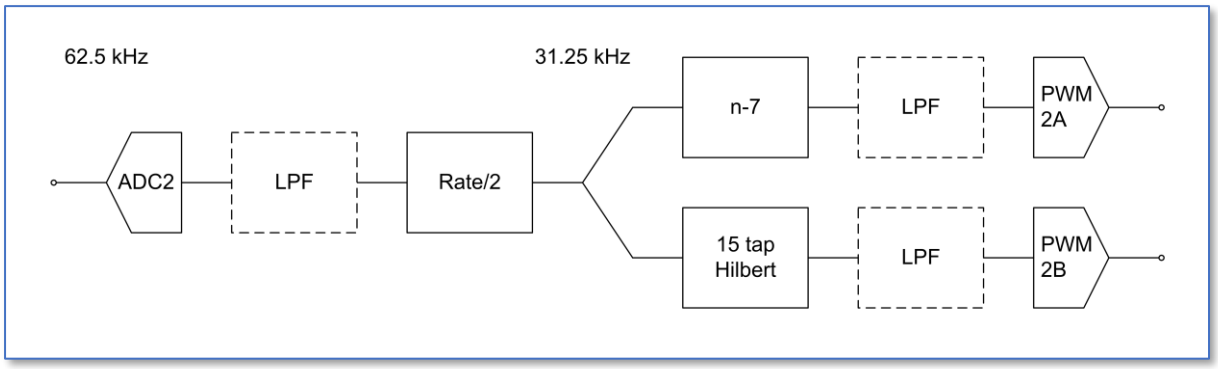
The coefficients for the taps can be derived from the Hilbert transform rules combined with the choice of a proper windowing function. The window function suppresses the ripple otherwise seen in the frequency response. See Iowa Hills tools to obtain a set of coefficients.

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. If this is the case in this project the sampling rate can be divided by 2, to 15.625 kHz.

Demodulation

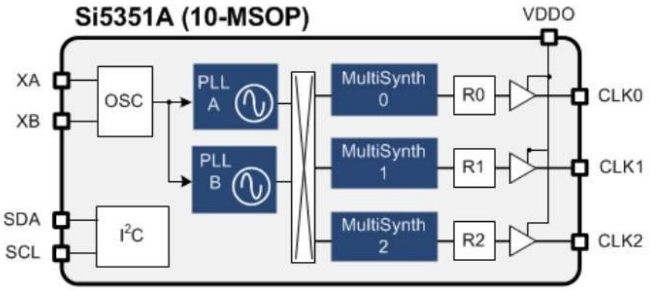
The I_{n-7} and Q_h (or I and Q) samples are used to obtain the desired audio signal. For SSB demodulation (USB) the I_{n-7} and Q_h samples are simply added.

Tx stream



The TX stream is the inverse of the RX stream: the same components can be found here as well. One difference is that the sample rate must be cut in half before doing the transformation, to obtain a similar frequency response.

Quadrature VFO (si5351.c)



The Si5351A is a triple clock generator, that can be controlled through I2C 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:

- 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 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

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.