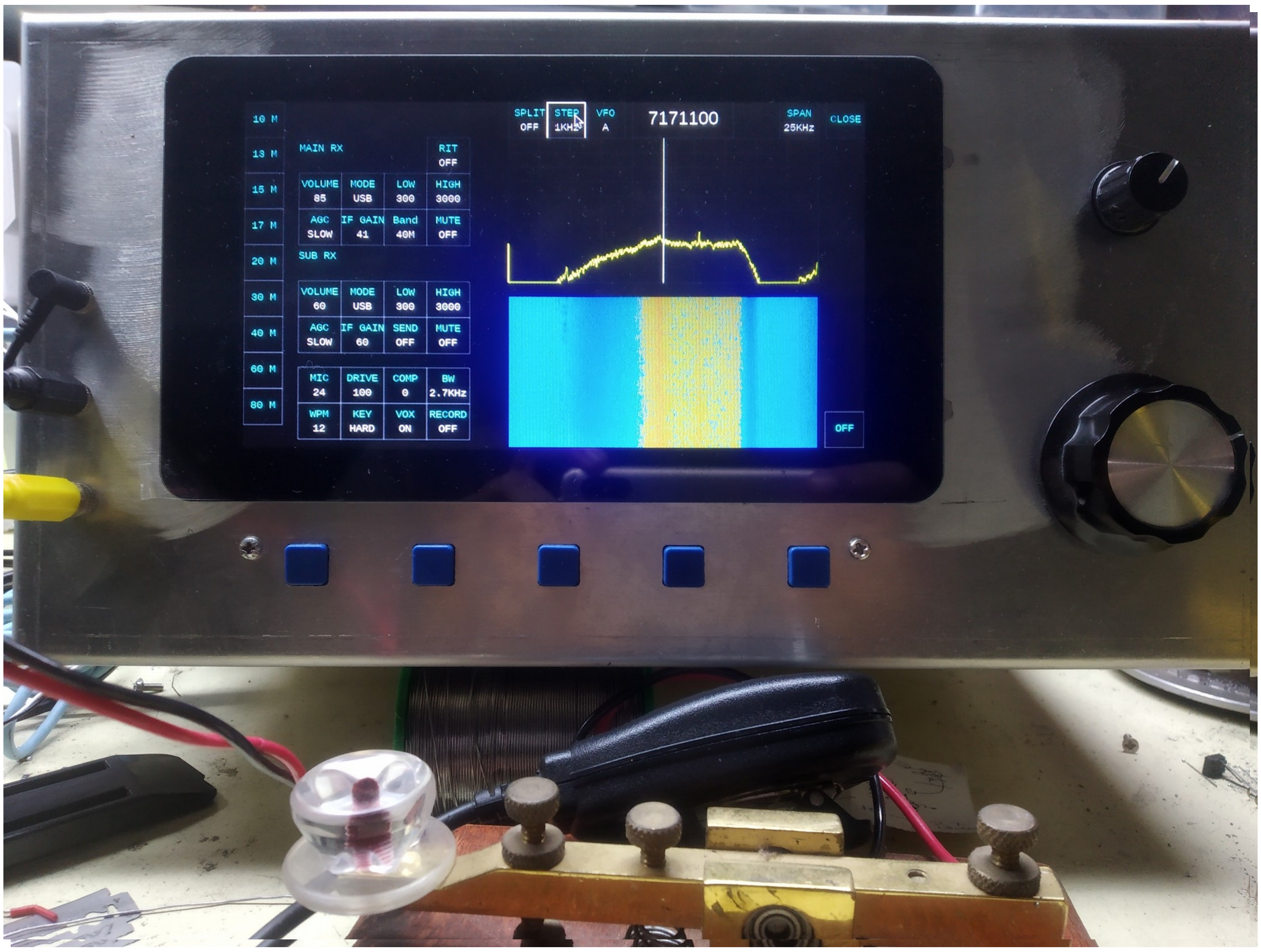
sBITX – An open source SDR that you can hack

- By Ashhar Farhan, VU2ESE



SPLIT OFF
STEP 1KHz
VFO A
7171100
SPAN 25KHz

MAIN RX
RIT OFF
VOLUME 85
MODE USB
LOW 300
HIGH 3000
AGC SLOW
IF GAIN 41
Band 40M
MUTE OFF

SUB RX
VOLUME 60
MODE USB
LOW 300
HIGH 3000
AGC SLOW
IF GAIN 60
SEND OFF
MUTE OFF

MIC 24
DRIVE 100
COMP 0
BW 2.7KHz
WPM 12
KEY HARD
VOX ON
RECORD OFF

Here how to convert your existing homebrew radio into a full-fledged SDR for less than $100.

The idea of hybrid architecture was first implemented in this form by Bob Larkin in his seminal work with DSP-10 transceiver. The details are in the ARRL book Experimental Methods in RF Design and its accompanying CD. An open secret about the EMRFD is that it is also an excellent introduction to Digital Methods and Bob is a great teacher. I spent a rainy day in Portland with Bob discussing radio and life and that evening we were joined by Jeff Damm (WA7MLH) for an early dinner. This project was born that evening.

Here are the headlines:

- Based on Raspberry Pi inside your radio
- Open source, plain C code that is easy to read, understand and change
- Adaptable to any superheterodyne radio that you may already have built
- Based on hybrid architecture
- Uses the simpler Fast Fourier Transforms in place of RF phasing systems

This article is divided into three parts:

1. **Our Game Plan**
   We will set out how we are going to build an SDR leveraging our skills with conventional superhet analog radios
2. **The Software**
   This describes the very core of the software, line by line. Transmit and Receive is broken down to make it easy to understand.
3. **SDR for the uBITX**
   We now adapt a standard QRP superhet (the uBITX) to become a full featured SDR.

Our Game Plan

There are many ways to build SDRs, (See the compact and informative presentation by Howard White, KY6LA at https://nparc.ca/wp-content/uploads/2019/06/Four-Generations-of-SDR.V3.0.pdf)

There are two standard approaches to building a software defined radio today:

1. **Direct Digital Conversion** : The radio directly digitizes the RF right at the front-end. For an HF radio, it can generate 60 million samples per second. Handling data at such speed needs specialized circuitry built using FPGAs (Field programmable gate arrays) in addition to a PC.

To build this kind of radio, you need multilayer boards, FPGA programming skills and also some money. This is clearly not what we want to do.

2. **Phasing Radio** : Two identical (called In-phase and Quadrature) direct conversion radios are used, both operated by the same local oscillator. However, the local oscillator drive to the Quadrature radio is delayed by exactly 90 degrees. The baseband audio from both the channels is fed to the two stereo inputs of an audio codec and the SDR software is run on a moderately fast CPU at audio frequencies.

The phasing radio has two main challenges. First, it needs a very precise 0 and 90 degree RF phase difference between the two channels. Second it also needs exactly similar audio gain and audio phase delays in both the channels. Even a 0.01 degree phase difference can degrade the opposite sideband suppression horribly. A nominal ladder filter made from 6 microprocessor grade crystals could do a far more efficient job than the most complex SDR front-ends. Every capacitor in the radio’s signal chain can change the phase and amplitude balance before it gets into the PC. Maintaining this over a large set of frequencies is a challenge especially if you are as sloppy a builder as I am.

Our approach is a hybrid one. On receive, using superhet architecture, we will bring out a 25 KHz slice of RF spectrum down to a low IF centered around 25 KHz (extending from 12.5 KHz to 37.5 KHz). A 24-bit audio codec running at 96000 samples/second will bring this digitized audio into the Raspberry Pi.

To transmit, we will generate the SSB/CW/FM/AM signal in software for a carrier centered around 25 KHz. We will upconvert this signal to the RF frequency of choice.

Engineering is the art of negotiation with science and economy. Our compromise is on limiting the maximum width of the waterfall to 25 Khz. Though this is not a major compromise, the experience shows that this waterfall is enough for the most. Rob Sherwood, NC0B, (known for his list of high performing radios) says that he prefers a 10 KHz wide spectrum when operating contests.

With this limitation in place, we can build an SDR that will run circles around those that cost thousands of dollars. This architecture was first implemented by Bob Larkin, W7PUA in his now famous DSP-10 transceiver it was copied by many commercial transceiver including the Elecraft K3 line, the FTDx-101D, etc. The links to his amazing series of articles are at the ARRL's SDR page on http://www.arrl.org/software-defined-radio.

The DSP-10 is almost a quarter century old design now and the sbitx is a tribute and a reboot of the original.

So here is the game plan. We start with a standard superhet design that you would have already built or bought as shown in *Figure 1.*

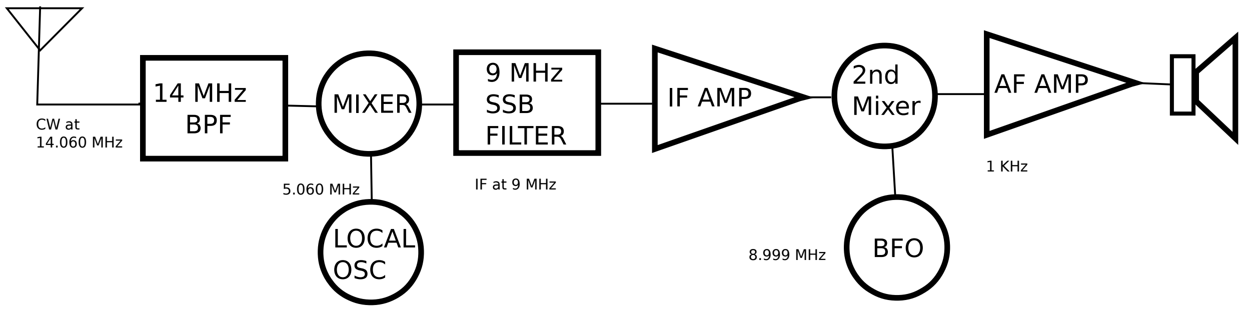


Figure 1

We will now modify this for the SDR.

1. Change the IF from 9 MHz to 27 Mhz in order to accomodate 25 KHz IF bandwidth. Most of our IF amplifiers are broadband and just changing the crystal filter will be enough. We choose 27 Mhz as the new IF frequency as we can build 25 Khz wide crystal filters at this frequency.

2. Change the local oscillator frequency to convert the incoming signal to 27 Mhz. To receive a 14 MHz signal, the local oscillator will have to now run at (27 + 14) 41 MHz.

3. The BFO is also moved to a new frequency such that it is 25 KHz away from the center of the crystal filtr. In our build, the center of the crystal filter passband was 27.005 Mhz, accordingly, the BFO was set to 27.030 MHz. With this BFO setting, the signals coming out of the 27 MHz filter are converted to a spectrum from 12.5 Khz to 37.5 KHz.

4. Open up the audio amplifier and remove any low pass filtering so that it handles frequencies up to 40 Khz.

5. Route the output of the audio preamp to a 24-bit, Wolfsun audio codec that

samples at 96 Khz. The audio codec is in turn controlled and connected to a Raspberry Pi 4.

6. The Raspberry Pi is loaded with the SDR software. It can also be loaded with WSJT-X, your favourite logger, fldigi and anything else you might wish for.

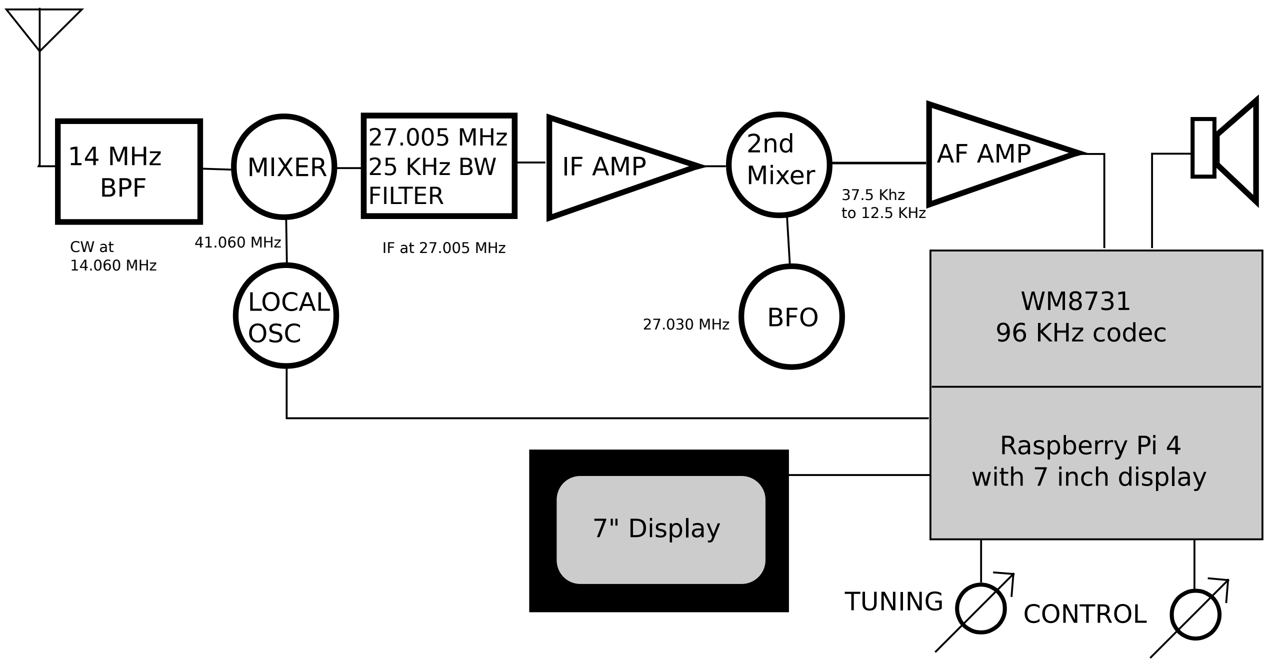


Figure 2

The modified superhet is presented in *Figure 2*. Let's examine it closely.

First thing that you will notice here is that the crystal filter is centered on 27.005 Mhz. All the ladder filters are essentially low pass filters and exhibit the center of the passband at a slightly lower than marked frequency of oscillations. You can consider the ladder filters as an evolution of low pass filters where the inductors are replaced by the crystals.

Microprocessor grade, low cost 27 Mhz fundamental mode crystals are now regularly available at very reasonable prices. We measured the motional inductance of the junk box 27 MHz crystals as 0.0038 H. A Min-Loss crystal filter was developed as shown in *Figure 3*.

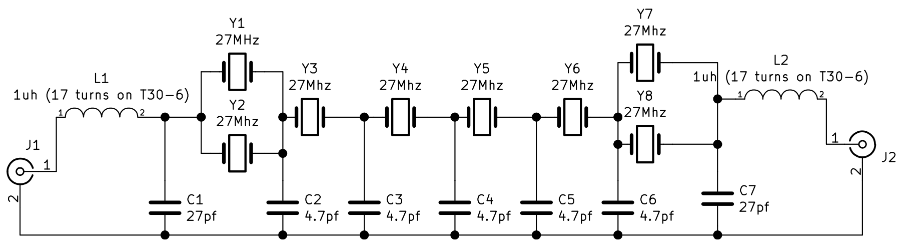
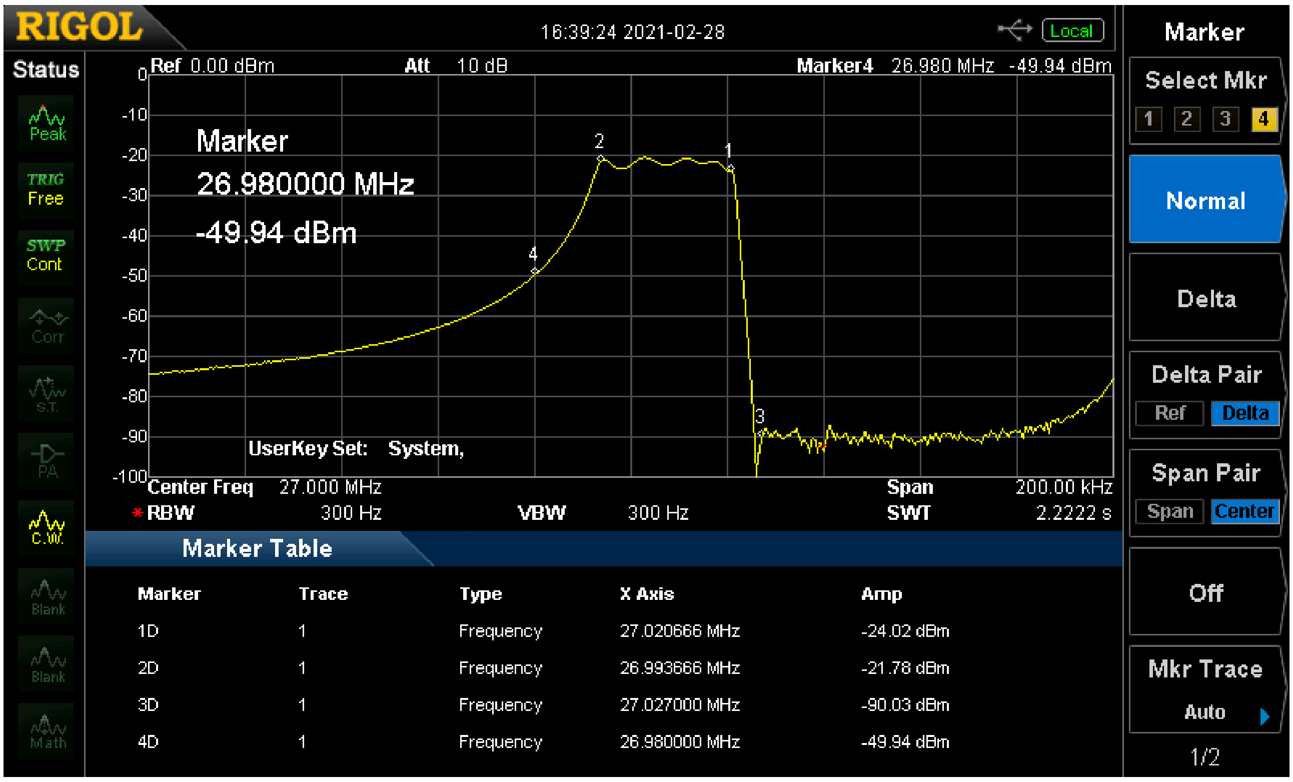


Figure 3



16:39:24 2021-02-28

Ref 0.00 dBm   Att 10 dB   Marker4 26.980 MHz -49.94 dBm

Marker
26.980000 MHz
-49.94 dBm

UserKey Set: System,

Center Freq 27.000 MHz   Span 200.00 kHz
RBW 300 Hz   VBW 300 Hz   SWT 2.2222 s

Marker Table

Marker	Trace	Type	X Axis	Amp
1D	1	Frequency	27.020666 MHz	-24.02 dBm
2D	1	Frequency	26.993666 MHz	-21.78 dBm
3D	1	Frequency	27.027000 MHz	-90.03 dBm
4D	1	Frequency	26.980000 MHz	-49.94 dBm

Figure 4

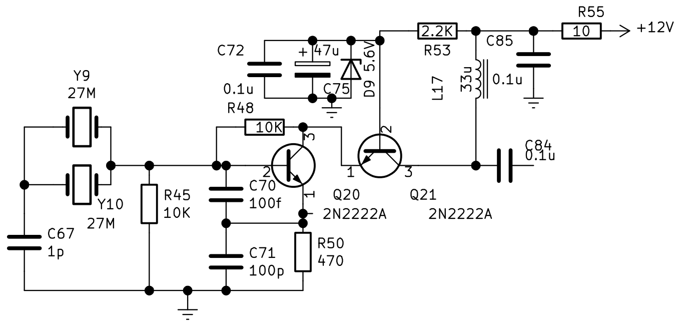


Figure 5

This crystal filter has a very steep response on the higher side. The measured response from the filter is in *Figure 4*. Sharp homebrewers will notice the filter ripple in the sweep image, it was eliminated after resoldering a capacitor). You will note that there is a very sharp attenuation of almost 70 db on the upper side (refer to Marker 4 in *Figure 4*). We must place our BFO above this frequency (27.030 Mhz in our case).

A BFO configured with two crystals in parallel and a 1pf capacitor in series can easily pull the frequency by the required kHz and yet exhibit nominal crystal stability. We have been able to replicate this BFO with many sets of crystals. A VXO candidate that we have built and tested is shown in *Figure 5*.

If your radio generates BFO from a programmable oscillator like the Si5351, then you wouldn't need to make a fresh BFO, you cans imply reprogram clock to a frequency between 27.027 and 27.031 Mhz, setting it 25 KHz away from the middle of your crystal's passband.

Let's now consider the WM8731 codec. The audio codecs produce and consume data at a very high rate. At 96,000 samples in two channels of 24 bits, every second, they have to transfer (96,000 x 2 x 24 = 4,608,000 bits per second) in each direction between the codec and the CPU. A serial bus called the I2S bus (not to be confused with the I2C!) is used to transfer data continuously between the CPU and the codec chip.

The I2S is a serial bus, and all the bits of each sample are sent one after another over a digital line. Two such digital lines are needed, one in each direction. The MISO line transfers data from the codec to the CPU and the MOSI line transfers the data from the CPU to the codec. The bits are transferred to the beat of the BCLK line on pin 3.

To maintain accurate timing, a crystal clock running at 12.288 Mhz is used along with the codec on pins 25 and 26. This crystal is a common crystal available from Mouser.com and other suppliers.

When data is being transferred serially at such speeds, it is easy to miss a bit and lose track of where each audio sample begins as all bits look the same. Hence, to mark when a left or a right channel data begins, yet another line is used that goes low on the left channel and high on the right channel. We configure the WM8731 to use the same clock for capture and playback, hence the pins 5 and 7 are tied together as the left/right clock (LRC). These four lines, the BCLK, MISO, MOSI, LRC, together handle the digitized data transfer between the codec and the Raspberry Pi. A two line interface of I2C is used to control codec parameters like volume, gain, etc. The code to initialize and use the codec is given in the repository at https://github.com/afarhan/sbitx/blob/main/sbitx_sound.c.

We hide away all the complexity of configuring the codec and the SDR programmer has to just handle a single function **sound_process().** This function is repeatedly called each time a new pair of 1024 samples arrive from the input of the codec. In turn, it returns back a pair of 1024 samples to be playback to the earphones / line-out.

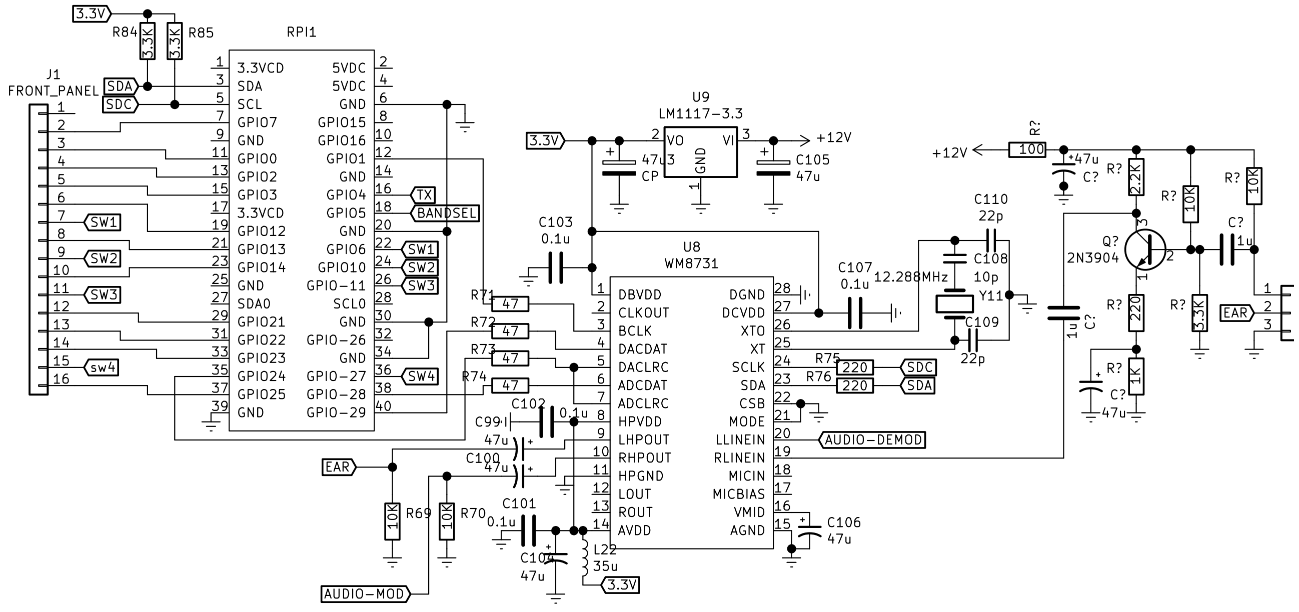


Figure 6

The Digital board piggybacks on a Raspberry Pi 4. Even an older Raspberry Pi could be used if you have one lying around, though an old RPi might not be able to decode WSJT-X in deep mode while simultaneously running the SDR. You can save yourself some trouble of buying and soldering WM8731 (it is an SMD, a large one as far as SMDs go) and just buy an assembled audio codec board from Mikroelektronika (https://www.mikroe.com/audio-codec-proto-board) for $19.00 USD. We highly recommend this approach.

There are a few subtle details in the digital codec board.

- The WM8731 has two grounds : The analog ground that is used as reference for the audio and the digital ground that is used as reference to the digital interface with the CPU. This keeps the digital noise of the Raspberry Pi isolated from the analog circuitry. If you are laying out your own PCB, keep the two grounds separate and connect them through thin tracks. Each side should have its own bypass capacitors. The WM8731 has its own linear regulator (U9). The codec consumes just a few milliamps of current and a linear regulator has far lower noise than a switching regulator. Read the WM8731 datasheets for more pointers.

- The WM8731 needs a 12.288 MHz crystal. They are available from mouser.com. When we first fired the codec, there was a 18 KHz constant carrier. It was later traced to fifth harmonic of the BFO (at 27.030 MHz x5 = 135.150 Mhz) beating with the 11th harmonic of the codec crystal (12.288 MHz x 11 = 135.168 Mhz) giving an 18 KHz beat. Adding a 10pf capacitor in series with the crystal pulls it up by 3 KHz to 12.293 Mhz, now the beat frequency will be (12.291 x 11 – 27.030 x 5 =)51 KHz, well outside the audio bandwidth of the 48 KHz. You may have to make this modification if you use an assembled board such as the one from Mikroelektronika.

- The Left Line-In channel takes in the signals now converted to a passband between 12.5 KHz and 37.5 KHz from the receiver, processes the signals in the RPi and plays the demodulated signal through the left earphone output.

- The Right Line-In channel takes the microphone audio, processes the audio in the RPi and plays out the modulated carrier at 24 KHz through the right earphone output to the modulator that raises it to 27 MHz.

- The radio is controlled through a number of digital lines. These can be used to switch between transmit/receiver, change band pass/low pass filters, etc. These are available on J2 connector

- The current front-panel has two rotary encoders and a 7 inch TFT capacitive touch screen (sold by Raspberry Pi). There are spare lines for a third encoder or three switches.

- The Raspberry Pi and the display together can consume upto 1.5 A of current at 5v. We used an LM2569 based buck converter from an online vendor. A pair of 33 uh inductors were added in series with input and output pins to eliminate switching power noise. The regulator is set to idle at 5.5V as the Raspberry Pi complains of low voltage if the power load brings down the 5v line ever so slightly.

The Raspberry is connected to the touch display through the DSI strip connector. The display needs a separate +5v power connection.

With the hardware architecture in place, we are now in a position to convert a 25 KHz slice of spectrum from the antenna to a low IF centered extending from 12.5 KHz to 37.5 KHz and digitizing it into the Raspberry Pi for signal processing. Similarly, on transmit, we will generate a signal between 12.5 KHz to 37.5 KHz and upconvert it to the RF frequency of choice.

The Software

Reading even the best of the texts on digital signal processing can make your head spin. if you do want to read, the freely available DSP Guide on www.dspguide.com is highly recommended.

Here, instead of discussing dry theory, we will take a radically different approach to understanding how SDRs work by strolling through the live code. The Table 1 is not pseudocode, it is the actual working receiver source code from our SDR! The code may change by the time you read this, but the general scheme will remain more or less the same. You can see the latest version of this code on https://github.com/afarhan/sbitx/blob/main/ubitx.c.

We will step through the function called **rx_process()**, the variables and other functions are all well described and documented in the rest of the source file, it is vital to understand the main routines of **rx_process()** and **tx_process()** and then look at how the other supporting functions help.

The kind of software method we use in our SDR is called *convolution filtering*. It is actually a simpler way to deal with signals compared to the normal approach of using writing FIR filters to introduce phase delays, etc.

Before we can understand the main routine, we need to grasp three concepts.

Concept 1: *The FFT spectrum is circular*

Let's have a working knowledge of the nature of Fourier Transform (FFT = Fast Fourier Transform). You can read more about this and download the code to perform them from www.fftw.org.

Our FFT routine converts the incoming signal samples into a spectrum and stores it in an array of bins held in **fft_out[]**. Each bin will hold the amplitude and phase of that bin's central frequency. For a bandwidth of 48 KHz that is represented in 1024 samples, each bin acts as a band pass filter of exactly (48000/1024=) 46.875 Hz.

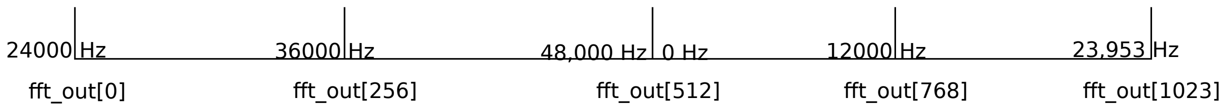


Figure 7

Consider the central frequency of 24 KHz as the zero point where our hypothetical BFO is. Frequencies above these are considered positive (The upper sideband) and those below are considered negative (The lower sideband). At **fft_out[0]** the values of amplitude and phase corresponds to 24000 Hz, from here the frequency increases and when we reach **fft_out [256],** it would have the values corresponding to a frequency of +36 KHz and so on until **fft_out[511]** corresponds to 43,953 KHz (exactly one bin below 48,000 Hz).

From **fft_out[512]** onwards, a weirdness sets in, the bin at **fft_out[512]** corresponds to zero KHz! You can also think of zero KHz as being 24,000 Hz behind the central frequency of 24 KHz as well. From **fft_out[513]** onwards the frequency keeps increasing towards the central frequency with **fft_out[768]** corresponding to 12 KHz and by the time you reach the last bing of fft_out[1023], you just behind the central 24 Khz by 46 Hz.

Stare at *Figure 7* to understand this. Re-read the two paragraphs above to understand this arrangement if necessary.

If you want to see a really beautiful, animated explanation of complex frequencies, your best bet is to watch https://www.youtube.com/watch?v=r18Gi8lSkfM.

Concept 2: *In frequency domain, just multiply the spectrum with the filter shape*

A simplistic way to filter in frequency domain is to zero out all the values in the fft_out bins corresponding to those that are outside the passband and convert the frequency domain bins back to time domain with a reverse FFT.

This almost works, except that it doesn't work very well.Sine waves are supposed to oscillate continuously, forever. The block of samples we are feeding into the FFT starts and ends abruptly at the beginning and end of the 1024 samples. This results in a smearing of the frequencies across the bins. You need to include frequency bins adjacent to your passband as well.

The solution to this smearing is called *Windowing*. Instead of abrupt cutting-off of the sides of a filter, you gently taper off the amplitudes on both sides of the passband. *Figure*

8 shows this graphically.

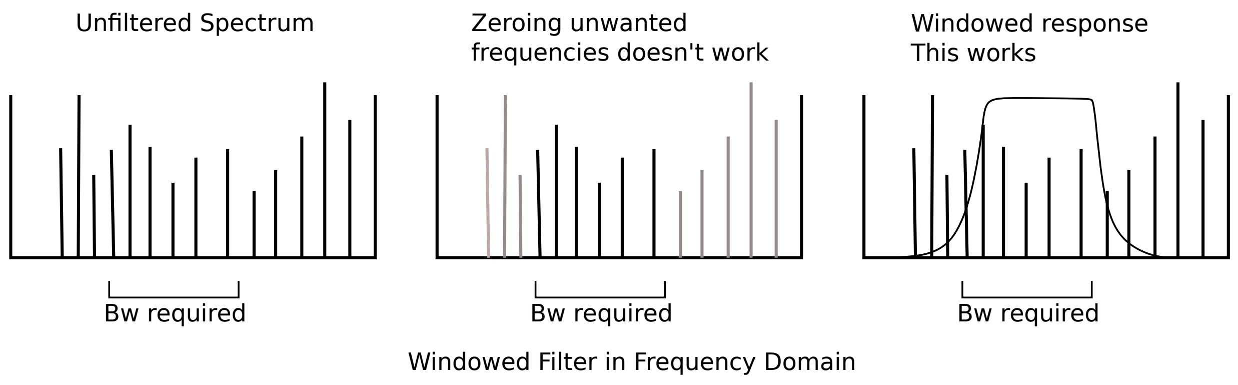

Unfiltered Spectrum
Zeroing unwanted frequencies doesn't work
Windowed response This works
Bw required
Bw required
Bw required
Windowed Filter in Frequency Domain

Figure 8

The filtering has to work in two steps.

In the first step, we generate the filter shape of the kind outlined in the right most graph of Figure 7. Thankfully, our mentor Phil Karn, KA9Q, has written these routines for us. They are in the fft_filter.c. These filters need to be generated just once for every bandwidth setting. There are various kinds of windowing functions and we use a simple one called the Kaiser window.

In the second step, the filter shape is simply multiplied with the spectrum. If a bin gets multiplied by 1, it goes through, if a bin is multiplied by zero, it gets killed, a bin multiplied by a coefficient between 0 and 1 gets attenuated.

Filtering in frequency domain is really simple. Just design your filter shape and multiply it with the frequency bins.

Concept 3: *Overlap and Save/Discard*

Consider that there are 1024 samples that represent a single continuous sine wave. When we perform an FFT over these samples, we expect to see a single pip in the spectrum corresponding to the frequency of the sine wave and zero everywhere else. However, when we actually notice the spectrum, it is not clean because the sine wave starts abruptly at the first sample. The abrupt start can litter noise across the spectrum (like a sudden burst of noise streaks the SDRs' waterfalls). This is a parallel of the problem we outlined in our Concept 2.

The solution is to perform FFT on 2048 samples instead of the 1024 samples. You take 1024 samples from a previous block, append the new block of 1024 more samples and perform FFT on this total of 2048 samples, bringing them into the frequency domain. We do all the frequency processing, and perform an inverse FFT to get back the 2048 processed time samples. The first 1024 samples will have the abruptness that we spoke about. The second set will be clean because the abruptness was 1024 samples ago and long smoothened out.

The original time samples are then stored away for the next iteration of processing. This scheme of processing to eliminate the noise is called the *Overlap-and-discard* or *Overlap-and-Save* scheme.

This is illustrated in Figure 9.

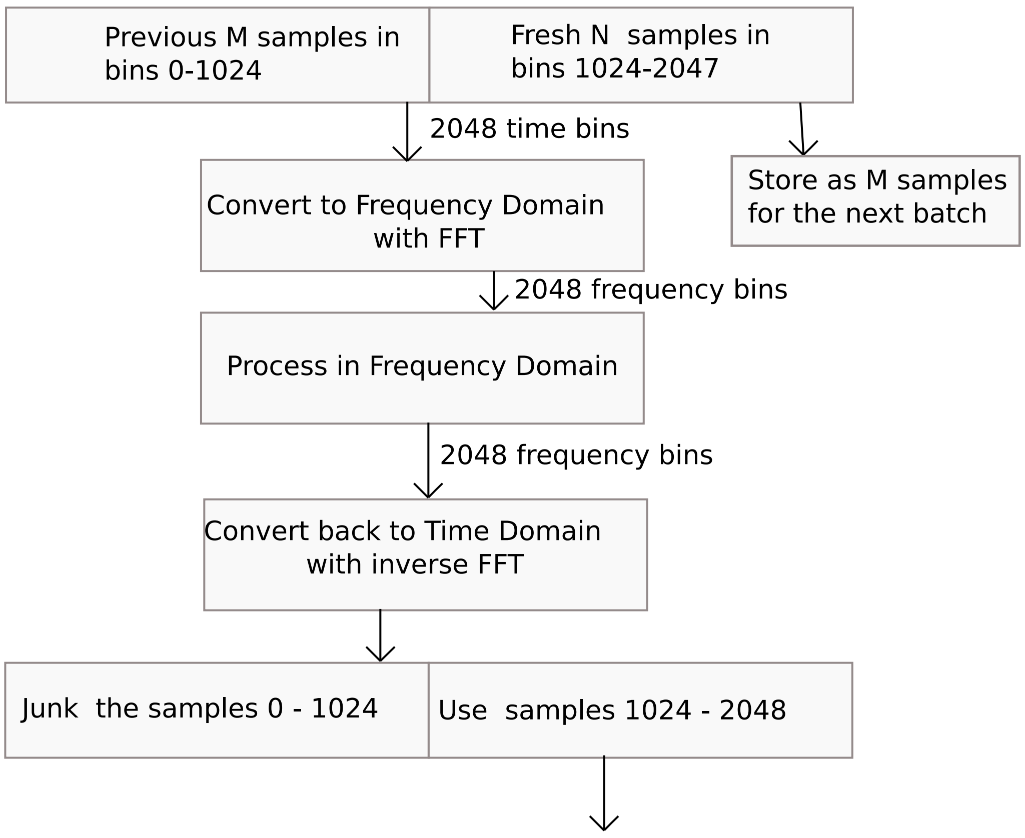

Previous M samples in bins 0-1024
Fresh N samples in bins 1024-2047
2048 time bins
Store as M samples for the next batch
Convert to Frequency Domain with FFT
2048 frequency bins
Process in Frequency Domain
2048 frequency bins
Convert back to Time Domain with inverse FFT
Junk the samples 0 - 1024
Use samples 1024 - 2048

Figure 9

The much promised walkthrough

With the three concepts cleared up, we can begin our code walkthrough. Refer to the Table 1.

Whenever 1024 new samples are ready from the codec, our sdr calls the **rx_process()**. The samples are passed in an array pointed to by input_rx, the parameter n_samples will hold the samples count (fixed at 1024 for now).

STEP 1

Our first order of business is to prepare for *Overlap-and-Discard*. We assemble 1024 samples into **fft_in[]** from the previous call to rx_process() in STEP 1. The previous samples are stored in an array called **fft_m**. It is callled **fft_m** because real FFT jocks call previous samples M, we follow.

Note that the samples received from the sound card were integer and these are scaled to floating point dividing them by 200,000,000.

STEP 2

While we append the fresh coming samples to the fft_in array, we notice something exciting - We are dealing with complex numbers! If this was an SDR that generated I and Q channels we would be assigning each of the I and Q samples to the real and imaginary components of each bin. However, in our case, the crystal filter takes care of the IF image and we filter the signals within the 25 KHz passband using the FFT, so we can safely zero the Q channel and write the incoming samples to the real part of the **fft_in[]** array bins. Notice that we also store the incoming samples back into the **fft_m[]** array for the next time **rx_process()** is called.

STEP 3

The fft is executed according to a 'plan'. In fftw library, the 'plan' is a preset collection of variables (like how many samples, how many dimensions) that have to be specified in each call to the FFT functions. Using a plan saves the hassle of having to specify it in every call. We initialize the fft plans in **fft_init()** function elsewhere in the same source file..

Our time samples are now converted into a spectrum with just one line of code by calling **fft_execute()**. The output of the **fft_execute()** are the frequency domain values; these are stored in **fft_out[]** array.

Wait, this is pure magic: we are barely on the 15th line of our SDR code and we already have a spectrum for display and waterfall!

The spectrum will have a little bit of smudginess that can be cleared by a four lines of code that I have removed to keep the distractions away from understanding the basic SDR flow. The spectrum display code can be read from the original source file.

STEP 4

The FFT output is organized as illustrated in *Figure 7*. When you want to bring in a signal at, let's say 10,000 KHz down to 0 Hz, in a direct conversion receiver we do it by mixing. In our convolution SDR, we simply shift the frequency bins around until the bins corresponding to 10,000 Khz is now at **fft_bin[512]**.

There is a very important "gotcha!". We said it has to be brought down to 0 Hz, not bin[0]. If you look at the Figure 7 again, you will see that the 0 Hz is at at **fft_out[512]** and not at **fft_out[0]**.

The variable **r->tuned_bin** holds the 'needle' of our radio's dial that points to the frequency that we want to convert to base-band. A simple loop shifts all the bins around by **r->tuned_bin** times.

STEP 5

We proceed to eliminate the other side band. Look at the *Figure 7* again. This is a picture worth a thousand words. Now, imagine that a signal that was at 10,000 Hz needed to be shifted to bin [512]. Accordingly, the signals that were below 10,000 Hz would have shifted to less bins below 512.

So, here is the catch : Once the bins are rotated, all the bins below 512 are lower sideband and those above 512 are upper sideband. To eliminate either side band, we simply zero it.

STEP 6

We now have our signal located starting at zero Hz, opposite sideband eliminated by brute force and all we have to do is to limit this signal to our desired pass band. The filter coefficients for the passband are already precalculated and stored in **r->filter->fir_coeff[]** array. We simply multiply the two arrays together.

To see how the FIR filter is calculated, read https://github.com/afarhan/sbitx/blob/main/fft_filter.c. Most of the code is written by Phil Karn, KA9Q. It is brilliantly simple. We must digress to understand how beautiful and simple it is.

Phil starts by taking an empty array of complex frequencies. THe sets it to '1' in those frequencies that should allow signals to pass through and zeros out those where the signal should be blocked. This brickwall filter is converted back to time domain by a call to the FFT routine. To the resulting time domain signal a windowing function is applied and converted back to the frequency domain by a second call to the FFT routine.

If you are curious, you can read how he does it in the **filter_tune()** and **window_filter()** from fft_filter.c.

STEP 7

We are done with everything now, all we have to do is apply the FFT again to shifted and filtered frequency domain bins of **r->fft_time[]** and convert them back to time domain. The last FFT call leaves the time samples in the **r->fft_time[]** array.

STEP 8

We apply AGC to the time domain signal . The AGC is not shown in this source code but is available in the ubitx.c. We have implemented the algorithm explained by Gerald Youngblood, K5SDR in his seminal series of papers (*ibid)*. It is a fairly simple function though at present, it generates a 'pop' on strong signals. We will have this debugged by the time you are reading this paper.

STEP 9

We finally ship the audio samples off to the earphones to be played back to the operator. Whew.

```
void rx_process(int32_t *input_rx,  int32_t *input_mic,
        int32_t *output_speaker, int32_t *output_tx, int n_samples)
{
        int i, j = 0;
        double i_sample, q_sample;

        //STEP 1: first add the previous M samples to
        for (i = 0; i < MAX_BINS/2; i++)
                fft_in[i]  = fft_m[i];

        //STEP 2: then add the new set of samples

        int m = 0;
        for (i= MAX_BINS/2; i < MAX_BINS; i++){
                i_sample = (1.0  *input_rx[j])/200000000.0;
                q_sample = 0;

                j++;

                __real__ fft_m[m] = i_sample;
                __imag__ fft_m[m] = q_sample;

                __real__ fft_in[i]  = i_sample;
                __imag__ fft_in[i]  = q_sample;
                m++;
        }

        // STEP 3: convert the time domain samples to  frequency domain
        fftw_execute(plan_fwd);

        //STEP 4: we rotate the bins around by r-tuned_bin

        struct rx *r = rx_list;

        for (i = 0; i < MAX_BINS; i++){
                int b =  i + r->tuned_bin;
                if (b >= MAX_BINS)
                        b = b - MAX_BINS;
                if (b < 0)
                        b = b + MAX_BINS;
                r->fft_freq[i] = fft_out[b];
        }

        // STEP 5:zero out the other sideband
        if (r->mode == MODE_LSB || r->mode == MODE_CWR)
                for (i = MAX_BINS/2; i < MAX_BINS; i++){
                        __real__ r->fft_freq[i] = 0;
                        __imag__ r->fft_freq[i] = 0;
                }
        else
                for (i = 0; i < MAX_BINS/2; i++){
                        __real__ r->fft_freq[i] = 0;
                        __imag__ r->fft_freq[i] = 0;
                }

        // STEP 6: apply the filter to the signal,
        // in frequency domain we just multiply the filter
        // coefficients with the frequency domain samples
        for (i = 0; i < MAX_BINS; i++)
                r->fft_freq[i] *= r->filter->fir_coeff[i];

        //STEP 7: convert back to time domain
        fftw_execute(r->plan_rev);

        //STEP 8 : AGC
        agc(r);

        //STEP 9: send the output back to where it needs to go
        if (rx_list->output == 0)
                for (i= 0; i < MAX_BINS/2; i++){
                        output_speaker[i] = cimag(r->fft_time[i+(MAX_BINS/2)]) * 1000000;
                        //keep transmit buffer empty
                        output_tx[i] = 0;
                }
}
```

The transmit process works exactly the same way, but in reverse. We will quickly step through it as well. If you have understood the rx_process(), the tx_process() is very similar, the steps are in reverse order. Like our analog radios, the SDR code can be bidirectional too!

The microphone input arrives at the right channel of the tx_process(), we take 1024 samples from the last call to tx_process, append another 1024 incoming samples to submit a total of 2048 samples to the fft_execute(). While inserting the incoming samples, they are scaled to become floating point values and they are backed up to be used in the overlap-and-discard process for the next call to tx_process() as well. This is the Overlap-and-Discard algorithm that we have already dealt with earlier.

The forward FFT converts these time samples into frequency bins.

The **tx_filter->fir_coefficients[]** are multiplied with **fft_out[]** to filter the signal to keep modulation bandwidth within limits. Here it must be stressed that for LSB, you have to apply a filter with the passband specified as (-3000 to -300) and for the upper side band, the filter is set to (300 to 3000). The lower sideband is made up of negative frequencies!

Next, the other sideband is eliminated by zeroing it. We now have the SSB signal available in **fft_out[]**. Except that it is centered at zero Hz.

We shift the signal up (for USB) or down (for LSB) the by **tx_shift** bins. With this shift, the signal is now at the correct transmit frequency within the 48 KHz bandwidth of the **fft_out[]**.

The last step is to convert the signal back to time domain with a second call to **fft_execute()** and copy the time samples to the output (right output channel of the sound codec).

```
void tx_process(
        int32_t *input_rx, int32_t *input_mic,
        int32_t *output_speaker, int32_t *output_tx,
        int n_samples)
{
        int i;
        double i_sample, q_sample;

        // we are reusing the rx structure
        struct rx *r = rx_list;

        //first add the previous M samples
        for (i = 0; i < MAX_BINS/2; i++)
                fft_in[i]  = fft_m[i];

        int m = 0;
        int j = 0;
        //gather the samples into a time domain array
        for (i= MAX_BINS/2; i < MAX_BINS; i++){

                i_sample = (1.0 * input_mic[j]) / 2000000000.0;
                q_sample = 0;

                j++;

                __real__ fft_m[m] = i_sample;
                __imag__ fft_m[m] = q_sample;

                __real__ fft_in[i]  = i_sample;
                __imag__ fft_in[i]  = q_sample;
                m++;
        }

        //convert to frequency
        fftw_execute(plan_fwd);

        // apply the filter
        for (i = 0; i < MAX_BINS; i++)
                fft_out[i] *= tx_filter->fir_coeff[i];

        if (r->mode == MODE_LSB || r->mode == MODE_CWR)
                // zero out the LSB
                for (i = MAX_BINS/2; i < MAX_BINS; i++){
                        __real__ fft_out[i] = 0;
                        __imag__ fft_out[i] = 0;
                }
        else
                // zero out the USB
                for (i = 0; i < MAX_BINS/2; i++){
                        __real__ fft_out[i] = 0;
                        __imag__ fft_out[i] = 0;
                }

        //now rotate to the tx_bin
        for (i = 0; i < MAX_BINS; i++){
                int b = i + tx_shift;
                if (b >= MAX_BINS)
                        b = b - MAX_BINS;
                if (b < 0)
                        b = b + MAX_BINS;
                r->fft_freq[b] = fft_out[i];
        }

        //convert back to time domain
        fftw_execute(r->plan_rev);

        //transmit output
        for (i= 0; i < MAX_BINS/2; i++){
                output_tx[i] = creal(rx_list->fft_time[i+(MAX_BINS/2)]) * volume;
                output_speaker[i] = 0;
        }
}
```

Together, **rx_process()** and **tx_process()** are each, a single page of code. It is pretty much amazing how simple an SDR can be.

We shall skip how the user interface works, how we implement CW (as a tone injected into the tx_process(), how the tuning works, etc. However, there are a number of things that one can appreciate here:

- Adding features like Audio Passband Tuning is just a matter of providing user with a way to generate a new set of coefficients for the **r->filter->fir_coeff[]**. You can experiment with different types of windowing functions (Blackman, Kaiser, etc.). Instead of trying to work the maths, just use the code and see what happens.
- The IF is at 25 KHz. You implement RX IF gain by multiplying (or dividing) the incoming samples by a ‘gain’ variable.
- Similarly, you can adjust the transmit gain by just multiplying the time samples in the last step of **tx_process()** to smoothen out the TX power output variation across different bands.
- Noise blanker/Noise reduction systems are yet to be implemented.

SDR for the uBITX

Note: The latest and more expanded version of building the SDR version of uBITX will be available on https://github.com/afarhan/sbitx . Potential builders should read the lastest and more comprehensive details in that repository.

With our understanding of the software and hardware complete, we can know apply this knowledge to modify a commonly homebrewed SSB superhet, the uBITX to become and SDR. This is just an example of how a conventional radio maybe adapted to become SDR. It will even have some features that are not present in any other SDR.

Reviewing the uBITX

The uBITX is a general coverage HF transceiver that can be built for $100 in new parts, you can read all the details of the uBITX on https://www.hfsignals.com/index.php/ubitx-v6/.

In Summary, the uBITX is a double conversion superhet. It uses a low pass filter to scoop in the HF Spectrum (0-30 MHz) and up-converts it to 45 MHz. A nominal, two pole 45 MHz crystal filter feeds to a second mixer that converts the 45 MHz signal to 11 MHz. An aggressive, 8 crystal SSB filter is used to allow only 2.4 KHz of bandwidth to pass through to a balanced diode demodulator to produce base-band audio. The audio is amplified by a single transistor before being applied to an LM386 to drive a speaker.

The uBITX uses termination insensitive amplifiers developed by Hayward and Kopski (W7ZOI and K3NHI) in the signal chain.

The transmit function reverses the direction of signal flow. The diode demodulator now acts as a modulator driven by a single transistor mic amplifier. The 11 Mhz double sideband signal is stripped of one sideband through the crystal filter, amplified and fed to the second mixer that up converts it to 45 MHz. The 45 MHz IF is then downconverted to HF output by the same mixer that was used to upconvert during receive operation.

Building the Digital Codec Board

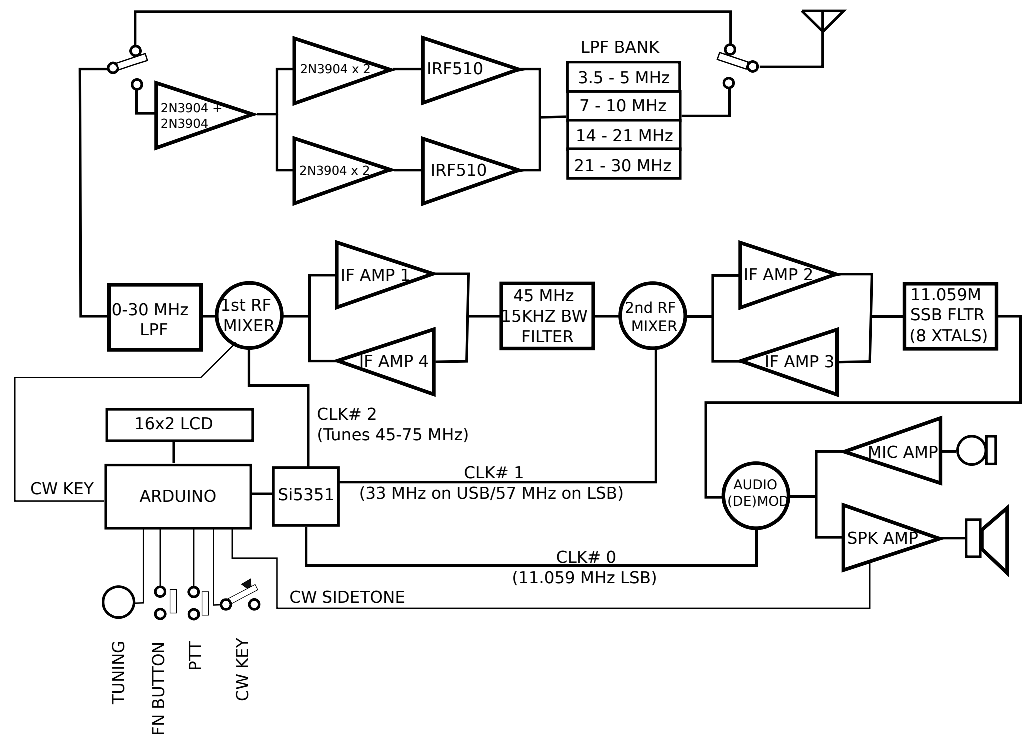
The digital codec board circuit is shown in *Figure 6*. You can build it on a pref board. The WM8731 codec IC is an SSOP chip wiht 28 pins. A commonly available SSOP-28-to-DIP adapter is used to mount the chip on the pref board. Alternatively, you can purchase the Mikroelectronika Audio Codec Proto board and mount it on the preboard as it has much of the audio circuitry of the digital codec board already pre soldered. It is available on mouser.com (https://www.mouser.in/ProductDetail/932-MIKROE-506) for $19.

If you are scratch-building the digital codec board, keep the analog and digital ground lines separate and join them only at the power supply point. Use extensive bypassing as close to the power pins as you can. Refer to the WM8731 datasheet for details (https://www.mouser.in/datasheet/2/76/WM8731_v4_9-1141834.pdf)

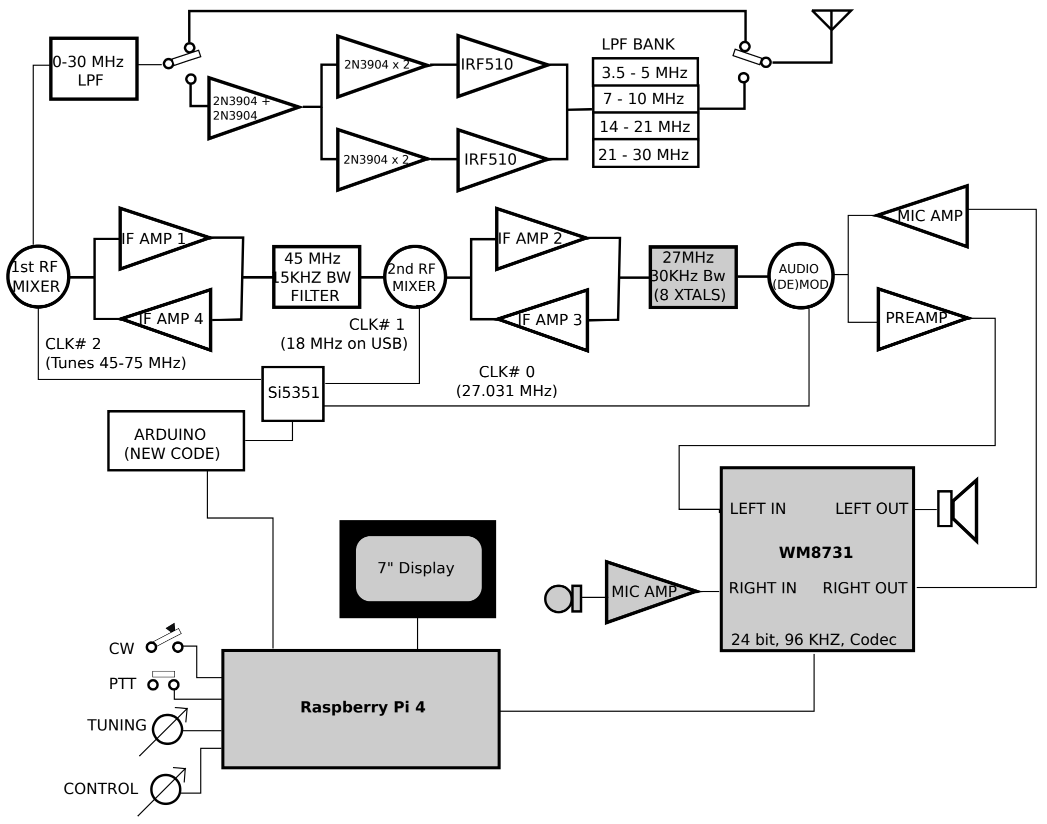
Modifying the uBITX

To software define the uBITX, some modifications have to be made to the original uBITX (all versions). The *Figure 10* shows the block diagram of the original uBITX and the modifications needed to make it software defined.

THE ORIGINAL uBITX



THE MODIFIED uBITX



The changes are :

1. Replace the uBITX's original 11.059 MHz crystal filter with 27 MHz filter shown in *Figure 3*.

2. Modify the uBITX microphone amplifier to reduce the gain. Remove the C61 and move C63 from Q6's collector to emitter. (*Figure 11*)

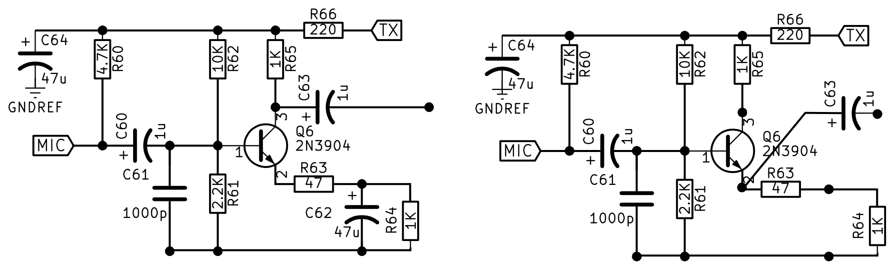


Figure 11

3. Connect the output from the hot-end of the uBITX volume control (AUDIO1 connector, pin 4) to the AUDIO-DEMOD line of the digital codec board (J2 connector, pin 13). Use a shielded cable to avoid noise pickup

4. Connect the input of the uBITX microphone (AUDIO1 connector, pin 1) to the AUDIO-MOD (J2 connector, pin 12). Use a shielded cable to avoid RF pick up on transmit.

5. Connect a USB cable from the Raspberry Pi to the Arduino mounted on the Raduino board of the uBITX.

6. Wire up the front panel with the two encoders, mic, ear, key from the J1 connector on the digital codec board as per *Figure 6*.

The *Figure 12* shows the overall wiring diagram.

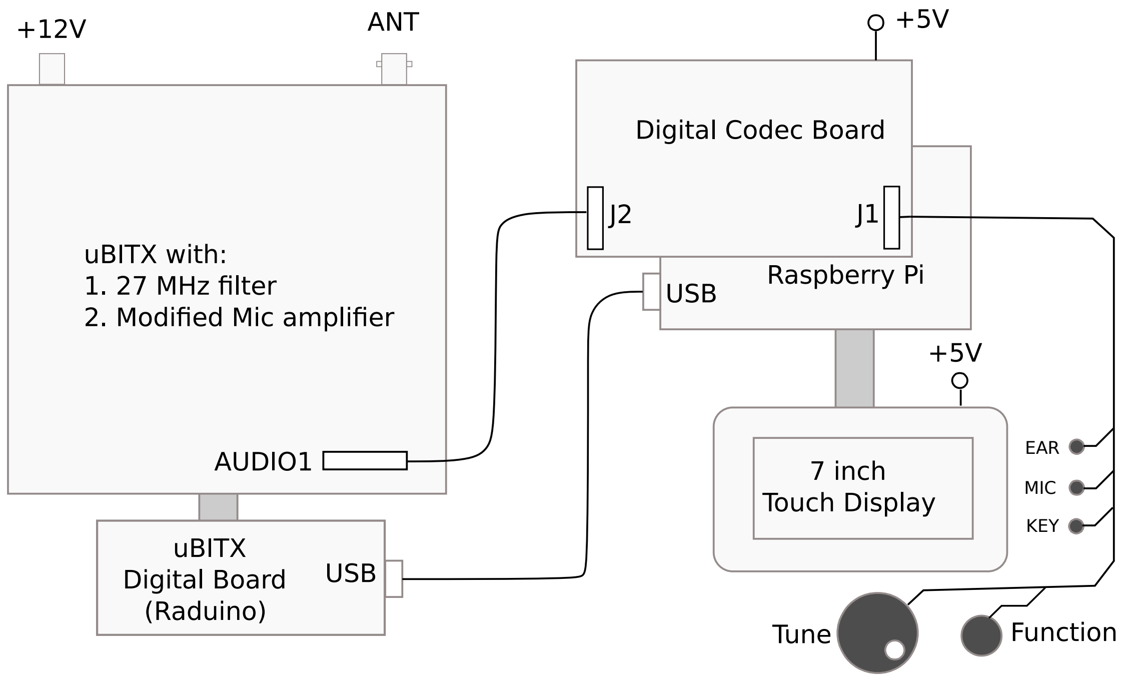


Figure 12

Installing the software

Before installing any software, calibrate the uBITX clock with the standard uBITX firmware.

There are two pieces of software we need.

1. Download the new Arduino sketch for uBITX SDR from https://github.com/afarhan/ubitx_sdr and upload it to the Raduino, the uBITX's digital board.

2. Follow the instructions on https://github.com/afarhan/sbitx to install the SDR on the Raspberry Pi.

It is recommended to install a shortcut to the SDR on the Program Menu of the desktop to start it without needing a keyboard/mouse.

The digital codec board (*Figure 6*) can be assembled on a general purpose PCB. The codec chip WM8731 can be soldered on an SSOP28-to-DIP adapter PCB and plugged into the 28 pin DIP socket on the perf PCB. Keep the digital and analog grounds separate.

It is highly recommended to power the digital codec board, the Raspberry Pi and the display adapter together from the Raspberry Pi's recommended power adapter. In the latest build, we power it up with an LM2569 buck regulator that was bought online. The regulator generated some switching noise which was cured by adding 33uh inductors on the regulator's input and output power lines.

Using the SDR

The screen is laid out to work in a very simple way. To change the value of any control, touch it and turn the function encoder to change its value. Toggle switches (like the switching between VFOs or selecting a different band) work by a single touch.

The user interface was developed to be a compromise between physical buttons and a software based user interface.

Many of the features of the SDR are still under development and they will be complete by the time of presentation of this paper, expect some changes.

Conclusion

The aim of demystifying SDR and bringing it into the homebrew lab is very ambitious. It clearly demands that the radio builder adapts to the digital world.

Many of us use circuit blocks like the Si5351 PLL or the ADE-1 mixer without delving into their internals. One could take a similar approach to the SDRs. We didn't really dive into how the FFT does its magic we use it like it were a building block. This is a reasonable approach to take until something stops working and then you get back to the texts.

The component count of adding features to a software defined radio is zero. You could spin out ten different filters for your radio without heating up the soldering iron.

Conversely, a hybrid architecture like that described here provides a new way of building high performance radios where the traditional challenges of dynamic range, low phase noise oscillators are still paramount. Our analog skills are not outdated, they are now evolved to make use of the immensely powerful and inexpensive computing power at our disposal that provides flexible modulation and signal processing processing. Many (including the author) would consider the time spent on a well performing AGC to be not worth the effort of component count, etc. These are no longer valid arguments in the world of software defined radios.

In this project, I must acknowledge that my own understanding is nominal and incomplete. I have tried to get by with minimum theory and a lot of experimental work to understand how things work. I have had the fortune of being mentored by Bob Larkin, W7PUA and Phil Karn, KA9Q into the world of digital signal processing,

Indeed a substantial part of the code is due to Phil. You can download, read and use Phil's code from http://www.ka9q.net/ka9q-radio.tar.xz. It is will be a weekend well spent on education. His code is readable and easy to follow. It is textbook perfect. I have borrowed liberally and proudly from his KA9Q radio project. You should too.

I studied the four papers written by Gerald Youngblood, K5SDR (SDR for the masses, Parts 1 to 4), reading them over and over, slowly. The Paper 3 is the most compact introduction to SDRs written from one homebrewer to another. I met him at Dayton and it was a privilege to just hang around the Flexradio booth and listen to him.

I am indebted to Wes Hayward, W7ZOI for being the elmer that he has been to such a large fellowship of homebrewers. His amazingly precise and clear enunciations of principles and methods has made a whole generation of homebrewers out of people who would have been wanderer/gatherers in the radio-land. The crystal filter that forms the kernel of this project is due to his writings on filter design. They are available on **www.w7zoi.net**.

My local gang of Lamakaan Amateur Radio Club has inspired and helped me with this project. Sasi VU2XZ, Thomas VU2TJ, Amar, VU2AAP, Rahul VU3WJM, Venu VU2BVB and Anil VU3DXA contributed to building and testing many of these circuits, on-the-air tests, contribution of ideas, circuits, components. At LARC (Lamakaan Amateur Radio Club) we are fortunate enough to have a really good bunch of radio amateurs with curiosity and stamina for ambitious projects.

My really long and fruitful discussions with Raj VU2ZAP have immensely helped the project. His unusual clarity of concepts helped cut through the fog many a times.

I am indebted to Bill Meara, N2CQR for agreeing to review this paper.

It sounds odd to be writing a conclusion for a software defined radio project. The software is never complete, there is always room for another line of code. The challenge is to keep it simple, modular. This project will hopefully make you pick up the virtual soldering iron, your keyboard, and build something in software for your own radio.

References:

1. Experimental Methods in RF Design (Campbell, Hayward and Larkin), published by ARRL, now out of print. Chapters 10 and 11. A complete introduction to DSP and SDRs in textbook format with minimum theory and practical implementational details

2. The DSP-10: An All-Mode 2M Transceiver using DSP IF and PC Controlled Front-Panel by Bob Larkin, QST Sept-Nov 1999. The sBITX is based on this architecture. http://www.arrl.org/software-defined-radio.

3. A Software-Defined Radio for the Masses by Gerald Youngblood, AC5OG, QEX July 2002 to April 2003. The Paper III explains convolution SDRs. http://www.arrl.org/software-defined-radio.

4. The DSP Guide by Stephen W. Smith. An easily understood online textbook on Digital Signal Processing. www.dspguide.com

5. Phil Karn’s, KA9Q radio source code. http://www.ka9q.net/ka9q-radio.tar.xz

6. The uBITX, An All-band HF Transceiver by Ashhar Farhan https://www.hfsignals.com/index.php/ubitx-v6/

7. Fourier Transform, Fourier Series, and frequency spectrum by Eugene Khutoryansky. A very intuitive animation of Fourier Transforms. https://www.youtube.com/watch?v=r18Gi8lSkfM

8. Filter Design Papers by Wes Hayward, W7ZOI. These are a series of a papers not available in any journal. Some of them simplify the design process and others explain it with a bit of maths. http://w7zoi.net/qststuff.html (They are at the bottom of the page.)

9. The sBITX source repository at https://github.com/afarhan/sbitx and the corresponding firmware for the uBITX at https://github.com/afarhan/ubitx_sdr. Find the latest source and installation instructions.