ka9q-radio

Overview

Ka9q-radio is a collection of software defined radio (SDR) modules connected by the Internet Real Time Protocol (RTP) and IP multicasting. It is intended to facilitate experimentation and education, and to be easy to interface to data decoders and digital communication programs. It can also smoothly track satellite Doppler shift in an open-loop mode when fed velocity and acceleration information from an external program.

The ka9q-radio modules execute from the command line on either Linux or OSX. As yet there is no fancy graphical user interface. (This may come later, especially if someone volunteers to create one).

License
The License for this software is GPL3, as detailed [here](LICENSE).

Installation
Installation instructions are [here](INSTALLING.md).

Multicasting and the Real Time Protocol (RTP)

The ka9q-radio modules are designed for maximum versatility. Unlike shell pipelines or TCP/IP connections, ka9q-radio modules can be individually started and stopped at any time, and one can feed any number of others without prearrangement. They can run on the same computer or on different computers on a multicast-capable network (e.g., an Ethernet LAN).

RTP was originally designed for multicast use, though it is now used mainly in a unicast (point-to-point) mode for voice over IP (VoIP). It provides sequenced delivery, but like all real-time and multicast protocols it cannot use retransmissions to guarantee delivery. This distinguishes it from the Transmission Control Protocol (TCP) that can only be used in a point-to-point mode for data that can tolerate latency.

Ka9q-radio uses four RTP payload types: raw I/Q data with a custom receiver status header; mono or stereo 16-bit uncompressed PCM audio sampled at 48 kHz; and the new Opus codec.

It should be easy to add PCM/RTP input to any ham SDR program that uses a computer sound card to acquire receiver audio (e.g. WSJT-X, fldigi and many others). This would allow these programs to accept receive audio directly over the network from the ka9q 'radio' program, completely bypassing the computer's sound system and freeing it for other uses. This would be a much cleaner and more reliable interface as there would be no analog audio cables (real or "virtual"), "rigblaster" adapter boxes, audio ground loops, equalization problems, or tricky level adjustments! With multicasting, any number of programs can simultaneously process the same receiver output.

Parts of the ka9q-radio package are well suited to "turnkey" networked receiver applications such as receive-only APRS-to-Internet gateways and Broadcastify feeds. Service descriptions are provided for Linux udev and systemd to load the daemons at boot time.

Opus Compression and Audio Monitoring

Although PCM is best for digital decoders, it requires 800 kb/s for mono and 1.6 Mb/s for stereo. Much lower data rates suffice for human listening over WiFi or a remote Internet connection. I use the new Opus codec standardized by the Internet Engineering Task Force (IETF). An excellent open-source implementation is available, and it can be freely used without patent restrictions (unlike, e.g., ABME or AAC). Opus automatically switches algorithms as needed to do the best it can with whatever data rate it is given, from very high fidelity stereo audio at up to (an unnecessarily high) 510 kb/s down to monaural voice at only 6 kb/s. (VK5DGR's CODEC2 can handle even lower data rates, which is on my to-do list.) I typically use Opus at 32 kb/s, which is actually much higher than needed for communications-quality audio.

Popular media players like VLC support PCM and Opus so you can listen to ka9q-radio audio on many smartphones and tablets. Currently the player must receive audio multicasts, which usually means it must be on the same LAN as the computer(s) generating them. Remote unicast and tunneled multicast connectivity is on the to-do list.

The ka9q-radio package also includes 'monitor', a RTP audio player that can receive and mix several multicast audio streams to the local sound output. It has a novel 'pan' feature that uses gain and delay adjustments (<1 ms) to place each source at a user-chosen point in a stereo image. This makes it easier to distinguish multiple sources, e.g., in a round table. The 'monitor' program currently supports only Opus and PCM, though CODEC2 is again on the list.

Direct, low-latency access to multicast data on remote networks requires some form of IP multicast routing or tunneling that is not (yet) provided in this package. The remote multicast data you are most likely to want is audio, so here's a simple workaround for remote listening:

```
ssh remotesys 'pcmcat -2 pcm.vhf.mcast.local | opusenc --quiet --raw --bitrate 32 -
-' | play -q -
```

This remotely executes the 'pcmcat' command, which picks up the specified multicast group (which must be PCM audio with RTP types 10 or 11), forces the output to be stereo (which opusenc expects by default), and compresses it with the Opus codec to 32 kb/s. The compressed audio is then sent over the SSH channel to your local computer where the 'play' program decompresses and plays the audio. This requires the 'opus-tools' package on the remote system and the 'sox' package on the local system. Latency can be several seconds because of shell pipeline and TCP buffering.

There's no performance penalty to running Opus in stereo mode on mono data, so for simplicity the ka9q-radio package always uses Opus in this way.

The KA9Q 'radio' program

The heart of the ka9q-radio package is the program 'radio', an interactive general coverage receiver. It reads raw I/Q data multicast by a SDR front end, sending it tuning and gain adjustments as

needed. Radio's output is a mono or stereo PCM audio multicast stream.

If audio compression is needed, a separate 'opus' module transcodes PCM to Opus. I.e., it receives PCM, compresses it with Opus and multicasts it to a different IP address. This lets listeners and players select compressed or uncompressed audio by merely joining the corresponding multicast group.

The user interface is simple but powerful. It runs in text mode, which makes it easy and fast to run remotely. If desired, the interactive interface can be disabled entirely and all parameters given instead on the command line (e.g., in shell scripts).

Architecture

The 'radio' program accepts a generic I/Q (complex) sample stream multicast by a computer with a direct conversion ("zero IF") SDR front end. Many inexpensive SDR front ends are available, though at present only the AMSAT UK FUNcube Pro+ dongle (heretofore called the "FCD") is supported (with the 'funcube' module). The (mostly higher) sample rates produced by other receivers such as the SDRPlay and RTL-SDR are supported but only the FCD's 192 kHz sampling rate has actually been tested.

An I/Q stream at 192 kHz with 16 bit samples is a constant 6.5 Mb/s network load; this is no problem for modern Ethernet but it should be kept away from inexpensive WiFi base stations and public Internets. (More on problems with multicast and WiFi in the end notes).

Hardware Artifact Removal

Although most SDR front ends use direct conversion, the complete SDR front end/ka9q receiver combination actually forms a dual-conversion superhet: the first LO and mixer are in analog hardware and the second LO and mixer are in software. The first mixer, a quadrature (I/Q) type, produces a complex first IF centered on zero Hz and extending from -Fs/2 to +Fs/2, where Fs is the A/D sample rate. For the FCD, Fs = 192 kHz so the IF represents a "window" on the radio spectrum from 96 kHz below the first LO frequency to 96 kHz above. (The Nyquist or minimum sample rate for a complex signal is equal to the bandwidth; for a more familiar real signal, the Nyquist rate is twice the signal bandwidth.)

This is often called a "zero IF" architecture but is more accurately called a "low frequency IF". The actual first IF can be anywhere between -Fs/2 and +Fs/2 (including zero) provided it contains the entire signal bandwidth. The 10 kHz near each edge from a FCD is best avoided because of imperfect anti-alias filtering before the A/D converters, and I presume other SDR front ends have similar limitations.

All analog direct conversion SDR front ends produce a "DC spike" at 0 Hz from crosstalk from the first LO to the receiver input. On the FCD this spike is about -50 dBFS (decibels below full scale, i.e., A/D saturation). The first processing step in 'radio' completely removes it, but reciprocal mixing produces a "mound" of low frequency phase noise that cannot be removed. With the FCD this noise is only noticeable within a few kilohertz of zero and then only when the antenna is disconnected to remove external noise. But it is still easy

to avoid by simply shifting the signal to a higher IF. It is common in these designs to place the IF near one quarter of the sample rate, e.g., the range from +48 kHz to +51 kHz (or -51 kHz to -48 kHz) might be selected for 3 kHz wide SSB. The 'radio' program tries to keep the IF in one of these two ranges as the user changes frequency.

48 kHz is still an extremely low first IF by usual standards, as ordinary pre-mixer filters cannot reject an image only 96 kHz away. This problem is avoided by the use of complex I/Q signals that permit the frequency image to be canceled with digital signal processing. Good image rejection requires the 'radio' program to compensate for any slight gain and phase imbalances between the I and Q channels, and this is its second processing step. Typical values for the FCD are 0.07 dB of gain imbalance and 0.4 degrees of phase error, i.e., deviation from exactly 90 degrees between the two channels.

Hardware artifact removal was recently added to the 'funcube' and 'hackrf' modules. It is still present in 'radio' but is disabled by default.

Frequency conversion

The third step is to convert the first IF signal to baseband, i.e., a carrier frequency of 0 Hz. This is performed by the second (software) LO and mixer using complex arithmetic to suppress the unwanted image. The LO and mixer each require only one complex multiply per sample. No lookup tables are used, and sine and cosine calls are needed only when changing the LO frequency.

Another LO and mixer are optionally used for open-loop correction of satellite Doppler shift. The 'radio' program reads velocity and acceleration from an external program, calculates and applies the appropriate frequency offset and rate.

When Doppler steering is active, a second LO is added to sweep the frequency of the first, requiring only two more complex multiplies per sample. Sine and cosine calls are needed only when the frequency sweep rate is changed. The Doppler steering is so smooth and accurate that I have been able to copy CW from a LEO cubesat transmitting on 70cm. Even on a near-overhead path, the signal was rock stable, staying in a narrow CW filter. (Of course, this requires accurate orbital elements and current clock time. It might be possible to use observed frequency errors to correct a set of orbital elements.)

Filtering and Demodulation

The baseband signal is filtered with fast correlation. With complex signals, the passband can be asymmetric, e.g., +100 to +3000 Hz selects upper sideband (USB) and -100 to -3000 Hz selects LSB. A post-detection frequency shift is provided for CW; this is equivalent to re-tuning the radio while simultaneously sliding the filter to keep the signal at the same point in the passband. Modes other than SSB typically use filters centered on 0 Hz but this is not required; the edges can be individually varied, e.g. to suppress adjacent channel interference.

The filtered baseband signal is then fed to one of three demodulators: 'AM', 'FM' and 'Linear.'

AM Demodulator

The AM demodulator is an ordinary non-coherent envelope detector that simply measures the amplitude of the complex baseband signal, ignoring the phase. A squelch needs to be added to make it usable for aviation communications.

FM Demodulator

The FM demodulator handles narrow-band frequency or phase modulation by taking the phase of a baseband signal with the arctangent function and then differentiating it (frequency is the derivative of phase). The arctangent instruction is surprisingly fast even on a Raspberry Pi, so a lot of literature on fast arctangent approximations seems to be unnecessary.

Two submodes are provided. 'FM' has a high pass cut on the received audio below 300 Hz to remove PL/CTCSS tones and a standard -6 dB/octave audio de-emphasis above 300 Hz. This seems to be common amateur NBFM practice.

The 'FMF' (FM Flat) mode is straight FM without post-detection filtering so the signal is treble-heavy and any PL tones are audible. Both modes use an FFT to measure PL/CTCSS tone frequencies to 0.1 Hz accuracy. SNR, frequency offset and peak deviation are also measured and displayed.

A classic FM demodulator ignores signal amplitude, but I use it in an experimental threshold extension scheme. Complex baseband samples with amplitudes falling below a certain fraction of the average signal strength are blanked. (The amplitude threshold is empirically set at 0.55 of the average). This is very effective at removing "popcorn" noise as the SNR decreases to threshold. I'd like to see how it compares to standard threshold extension techniques such as a narrow PLL.

The FM squelch works from first principles: the signal-to-noise ratio is estimated and the squelch opens when it exceeds +6 dB. This works so well that I've felt no need for a squelch adjustment. The squelch originally closed so quickly that there was no tail, but I added a short one to ensure that the tail end of a packet transmission isn't cut off.

After the squelch closes the silent PCM output (containing all 0's) continues for a few packets to flush any digital filters decoding the stream. The PCM audio stream then stops to reduce network load. The Opus transcoder, if used, also stops producing compressed packets. When the squelch reopens, the RTP marker bit is set to flag a jump in the sample count. The RTP timestamp indicates how many silent samples were suppressed. This allows programs using the input stream for timing to maintain a proper sample count.

Linear Demodulation

The 'linear' detector is used for all other modes, including SSB/CW, ISB (independent sideband) and raw I/Q. In linear mode the filter output is the audio output: both the I and Q channels for stereo modes and just the I channel for mono. Stereo is implied by I/Q and ISB. It

is optional for CW/SSB, where it produces an interesting effect on headphones because of the 90 degree phase shift (Hilbert transform) between the left (I) and right (Q) channels -- the sound no longer seems to come from a point in the middle of your brain. I think this might be useful in decreasing fatigue during long contests.

The ISB mode is the same as raw I/Q except that the filter cross-adds the positive and negative spectral components to put the lower sideband on the I (left) channel and the upper sideband on Q (right).

A PLL can coherently track an AM carrier for synchronous detection, which is usually much less noisy than regular AM envelope detection. The carrier frequency offset is averaged and displayed; This is useful for calibration of the SDR TCXO against an external reference such as WWV/WWVH or the pilot carrier of a local ATSC digital TV transmitter.

A squarer can be enabled so the PLL can track the suppressed carrier in DSB-AM and BPSK.

User Interface

The 'radio' program has a textual user interface that uses the "ncurses" library. It looks primitive by 2018 standards, but it is fully functional and very efficient over slow Internet paths. The `h` key pops up a summary of commands. The textual user interface can also be completely disabled and all receiver parameters given on the command line, e.g, for invocation from a shell script in a dedicated system.

Startup parameters are stored as files in the .radiostate directory in the user's home directory. The parameters include the input and output multicast domain names (or IP addresses), the tuning frequency and step size, the filter settings and detector mode. The correlator block size and FIR filter length can also be set here; note they cannot be changed after the program starts except by stopping and restarting it.

Invoking 'radio' with the name of a startup file loads its settings. The state of the radio can be saved to a state file with the `w` key. On normal termination the radio state is also stored as "default". It will be automatically loaded on the next invocation if no explicit name is given.

The various operating modes ("AM", "USB", etc) are defined in /usr/local/share/ka9q-radio/modes.txt. Each entry specifies a demodulator along with filter parameters, the post-detection frequency shift, AGC responses and option flags. For example, "USB" selects the "linear" demodulator, a filter passband of +100 to +3000 Hz, a zero post-detection frequency offset, an AGC attack rate of -50 dB/sec, an AGC recovery rate of +6 dB/sec, an AGC hang time of 1.1 seconds and mono output.

The textual screen display consists of the following status windows:

The "Tuning" window shows the RF, LO and IF frequencies. All can be changed by the user. If Doppler correction is active, it is also shown here but cannot be modified by the user.

The user can change frequency with the arrow keys, a Griffin Power Mate USB knob, a mouse wheel, or by typing a complete frequency into the pop up window invoked with the `f` command. The cursor indicates the digit that will change with the USB knob, mouse wheel or up and down arrow keys. The cursor can be moved to a different digit with the left and right arrow keys, and it can be moved to the next field with the TAB key. (Shift-TAB moves to the previous field.)

The "Carrier" and "Center" fields usually differ only in the linear modes when the filter edges are not symmetric around 0 Hz. The "Carrier" field shows the (suppressed) carrier frequency that comes out at zero Hz in the receiver output while the "Center" field indicates the center of the receiver passband. For example, in USB mode with a +100 to +3000 Hz frequency response, the "Center" frequency will be (100 + 3000)/2 = 1550 Hz above the carrier frequency.

The "shift" setting shifts the entire audio output spectrum; the indicated carrier frequency appears at the selected frequency in the audio output rather than 0 Hz. This can be useful for CW operation but is not required.

The radio is most easily tuned through the "Carrier" or "Center" fields, but it can also be indirectly tuned through the First LO and IF fields. The displayed LO frequency includes the estimated TCXO frequency error and the effects of fractional-N frequency synthesis (see the end notes).

Any number of copies of 'radio' can process the same I/Q stream provided they share the same frequency range. When there is a conflict, the last radio to re-tune the SDR LO wins. The other copies will not try to fight it; without further user input they will simply wait until the LO is again within range. To avoid unintentional changes in the LO frequency, tab to the First LO field and hit the `l` key; this will underline the entry to indicate that the field is locked. The Carrier/Center frequency fields can also be locked.

The LO frequency in an I/Q recording is set at recording time and cannot be changed during playback.

The "Signal" window shows signal, noise and SNR estimates at various points in the signal path. Levels are in dBFS, decibels below full scale (A/D saturation) as there is no way to know the analog gain ahead of the A/D converters without careful calibration, which is invariably frequency dependent. These are all output-only fields.

The "Info" window shows available information on the name of the tuned channel or frequency band. For a ham band, the allowed emissions and required license class are given. These are all output-only fields, taken from /usr/local/share/ka9q-radio/bandplan.txt.

The "Filtering" window shows the pre-detection filter settings. The first four parameters are user-adjustable as part of the field sequence selected with the TAB key. Smaller Kaiser betas give sharper filters but poorer stop-band attenuation; 0 is equivalent to rectangular windowing, i.e., no windowing at all. Larger betas give wider transitions but better ultimate rejection outside the passband. The block size and FIR (which determine the FFT frequency bin size and

filter delay) can only be set at startup from a state file or on the command line. [See the later discussion about sample rates and decimation.]

The "Demodulator" window is mode-specific and output-only. The AM and linear modes include a simple AGC and the relative audio gain is shown (it needs a real S-meter). The FM mode shows the frequency offset, peak deviation and PL/CTCSS tone (if present). When the selected mode and options permit the signal-to-noise ratio to be estimated, it is shown. (These estimates are done differently from the SNR estimate in the "Signal" window so they may not exactly match.)

The "Options" window selects the various options for the "Linear" demodulator. You can select an entry with the mouse or type the mode into a pop up box with the `o` command.

The output-only "SDR Hardware" window shows the nominal A/D sample rate, TCXO offset, nominal tuner frequency (uncorrected by the TCXO offset estimate), analog gain settings and gain and phase imbalance estimates.

The "Modes" window allows a predefined operating mode to be conveniently selected with the mouse.

The "I/O" window gives information on the input and output multicast network streams: IP addresses or host names; port numbers; packet, sample and error counts; and a SDR timestamp useful when playing back recorded I/Q data. It also estimates the true input A/D sample rate against the local computer time-of-day clock.

A "Debug" area at the very bottom of the screen shows version information and various test and debugging messages.

Textual user input entered through pop up menus; for example the `f` key pops up a box into which a frequency can be directly entered as `147m435` (147.435 MHz), `760k` (760 kHz) or `1g296296` (1296.296 MHz). If no letters are used to indicate the magnitude of the decimal point, a 'best guess' is made to produce a valid frequency; e.g. 14313 will give 14.313 MHz, not 14.313 kHz (which is below the FCD's range). Note that some ranges are inherently ambiguous; e.g., `500` could be either 500 kHz or 500 MHz so they should be typed with explicit decimal points.

'radio' in quiet/daemon mode

The 'radio' program can run in a totally 'quiet' mode with no user interface. This is suitable for automatic execution at boot time for fixed channel operation (e.g., receiving and reporting APRS transmissions). Two Linux systemd service descriptions are provided: 'radio34' and 'radio39'; the first creates an instance of 'radio' listening on 144.39 MHz in FM mode and the latter does the same on 144.39 MHz. Thise two instances can share the same Funcube dongle because their frequencies are only 50 kHz apart, within the 192 kHz bandwidth of the Funcube dongle.

144.39 MHz is the standard APRS frequency for North America; 144.34 MHz is the secondary APRS frequency used by WB8ELK's 15-gram 'pico tracker' for long duration balloon flights.

Other ka9q-radio Modules

Other modules in the package provide miscellaneous functions and/or start more complex planned features. None have especially complex user interfaces; most take only a few command line arguments and can run as background daemons. They are given in alphabetical order

aprs

This unfinished module accepts decoded AX.25 frames from the 'packet' module, extracts APRS position data from a selected station, computes the azimuth, elevation and range to that station from a specified point, and commands antenna rotors to point at the transmitting station. This was written specifically for tracking high altitude balloons.

aprsfeed

This module also accepts decoded AX.25 frames from the 'packet' module. It feeds those frames to the international APRS network so position reports will automatically appear on sites such as www.aprs.fi. It can be run as a background daemon.

funcube

This module takes the digital IF from a locally connected FCD and multicasts it over the local LAN. It accepts unicast commands to tune the radio and set analog gains. It will automatically reduce gain to avoid A/D saturation. Similar modules will be written for other SDR front ends such as the SDRPlay and RTL-SDR that will either talk directly to the hardware or through "shimware" such as SoapySDR.

The 'funcube' module can be automatically loaded at boot time (or at device insertion) as a background daemon by the Linux udev/systemd subsystem. A udev rule file and a systemd service file are provided. Note that the latter specifies the multicast group for the digital IF data, and this will probably require local configuration.

In daemon mode, 'funcube' writes its status to /run/funcube#/status, where '#' is the device number (starting at 0). Scripts for two devices are provided. The Raspberry Pi can support two devices, but I prefer one Pi per dongle when possible.

hackrf

This module is functionally equivalent to'funcube' except for the HackRF One. Only reception is currently supported. The HackRF supports sample rates up to 20 MHz but has only a 8-bit A/D, so by default it samples at a high speed and decimates to 192 kHz, the same as the funcube. The USB data rate and decimation processing load is too great for a Raspberry Pi, but it will run on a low-end x86 system.

Like 'funcube', 'hackrf' can be automatically loaded at boot time under Linux.

In daemon mode, 'hackrf' writes its status to /run/hackrf0/status. (Only one hackrf device is currently supported, but this can be changed.)

iqrecord and iqplay

These modules perform the functions suggested by their names. 'iqrecord' accepts multicast I/Q or PCM audio streams and writes them to disk. Gaps in the multicast stream due to lost packets or silence suppression are seeked over in a recording to maintain the correct sample count and playback timing. With a file system that supports "holes", disk blocks need not be allocated to these silent periods.

Raw I/Q streams are written into files named as 'iqrecord-xxxxxx-n' where xxxxxx is the RTP SSRC (Stream Source Identifier) and 'n' is a number incremented to avoid overwriting existing recordings. PCM streams are written as 'pcmrecord-xxxxxx-n'. Both files are written as raw, header-less 16-bit PCM with all meta information in extended file attributes.

Most but not all modern file systems support extended attributes; a notable exception is Microsoft's VFAT32 often used as a least-common-denominator for file exchange between different operating systems. Warning! Many file copy and archiving utilities do not preserve extended attributes, at least not by default.

I/Q and PCM recordings have many uses. An I/Q recording preserves everything within the front end passband regardless of modulation type or bandwidth (provided it fits within the receiver passband). This is useful for testing and for demonstrating the 'radio' program when an antenna is not available.

I/Q recordings are especially useful as backups during critical events, such as a satellite pass or balloon flight. An I/Q recorder placed next to the SDR hardware depends only on the antenna, SDR hardware and I/Q multicast program (e.g. 'funcube'). This can guard against bugs (or the outright failure) of downstream network or processing elements, e.g., a digital data demodulator. After the demodulator is fixed, the recording can be played back to recover the data.

And of course nothing limits you to just one I/Q recording.

'iqrecord' is an excellent example of the versatility of multicasting; it can just sit quietly in the corner backing everything up without impairing any other element that might be processing the same stream.

The 'iqplay' module plays back I/Q recordings (only -- no PCM support at present) using the meta data contained in the external file attributes. It can also read a raw I/Q sample stream from standard input to simulate SDR front end hardware.

modulate

A simple (and unfinished) test modulator that takes baseband audio, amplitude modulates it on a specified carrier frequency, and emits it on standard output as an I/Q sample stream.

opus

The 'opus' module was described earlier as an optional "transcoder" that accepts uncompressed PCM (either mono or stereo) and produces a lower bit rate compressed version with the Opus codec. Options include

a range of block sizes from 2.5 milliseconds to 120 ms. The larger blocks are useful only to reduce packet rate and header overhead; they provide no additional compression and are supported only by more recent versions of the Opus library. The default is 20 ms, which is long enough to give good compression but short enough to be essentially unnoticeable.

The compressed output data rate is specified by a command line parameter that defaults to 32 kb/s. This produces surprisingly good audio even on stereo music.

The Opus codec also supports a VOX-like "discontinuous" mode well suited to half-duplex push-to-talk amateur operation. When silence is detected, it automatically drops to sending "comfort noise" at only 3 packets per second even as the PCM input stream continues. If the PCM input stream stops, the Opus output stops regardless of the discontinuous setting.

Opus streams are always stereo even when the audio is mono. There is no capacity penalty and it simplifies things.

'Opus' can be run as a daemon; systemd 'service' config files are provided.

opussend

This is a standalone utility that takes PCM audio from a local sound interface, compresses it with Opus and multicasts it as an RTP network stream.

packet

This module is my first digital demodulator module for the ka9q-radio package. It accepts PCM audio from the 'radio' program, demodulates the ancient Bell 202A AFSK modem tones and decodes AX.25 frames. The decoded frames are multicast as RTP/UDP packets. The decoded frames can also optionally be displayed on the console. Otherwise the module can run as an unattended daemon.

'Packet' can be run as a daemon; a systemd 'service' file is provided.

pcmsend

This is the same as opussend, except that the output stream is uncompressed 48 kHz PCM (mono or stereo).

pcmcat

This joins a specified multicast group, which must carry uncompressed PCM audio (RTP types 10 or 11) and emits the PCM stream on standard output. This is useful for piping into an audio compressor for remote transmission.

Footnotes and Side bars

Sample Rates and Decimation

The I/Q input sample rate must be an integer multiple of the 48 kHz audio output rate. Decimation is performed as a byproduct of the fast correlator used for pre-detection filtering. Since the input FFT in a

fast correlator executes at the input sample rate, CPU loading will increase. Although the CPU loading at 192 kHz is small even on a Raspberry Pi, faster and simpler decimators are in the works to support much higher sampling rates.

By default the filter decimates the FCD 192 kHz A/D input sample rate by a factor of 4 to a 48 kHz audio output. Other SDR front ends with higher sample rates will require correspondingly higher decimation ratios. Although 48 kHz may seem excessive for communications-grade audio, I don't recommend reducing it. 48 kHz is supported by nearly every audio D/A, and it still uses only 0.154% of a gigabit Ethernet link.

The Opus codec strongly prefers 48 kHz stereo even for narrow band mono voice, and there seems to be no advantage to mono or a lower sample rate (i.e., the compressed data rate is not reduced nor is audio quality improved at a given rate.) So if the audio bit rate is a concern, just use Opus with a specified data rate; don't reduce the PCM sample rate.

Because digital demodulators should be fed uncompressed PCM, they are best run on a computer with a fast network path to the radio program so they can be given uncompressed PCM. (Audio compression works by eliminating signal components inaudible to the human ear, but which may be very important for a digital demodulator.)

Filter Tradeoffs

Simultaneously improving both filter roll off and stop-band attenuation requires a longer FIR impulse response. This requires greater latency and somewhat increased CPU loading. Currently, the filter block size and FIR length can only be specified on the command line or in the startup file, i.e., they cannot be changed without restarting the program. Note: the length of the FFT executed by the filter is equal to the sum of the block size and FIR length minus one. The default block size of 3840 corresponds to 960 samples after 4:1 decimation to 48 kHz; this is the number of samples in a 20 ms Opus frame. 3840 + 4353 - 1 = 8192, i.e., the FFT has 8k points. While the FFTW package can handle any number of points, a power of 2 is more efficient. I've found the default parameters to be suitable for most purposes.

Fractional-N Frequency Synthesizer Artifacts

Because the first LO in the FCD is in analog hardware with the usual synthesizer tuning glitches, it is re tuned only when necessary to contain the desired signal in the first IF, i.e., between -Fs/2 and +Fs/2. Otherwise tuning is with the second (software) LO that can be smoothly and instantly tuned without glitching. This is especially important when decoding digital satellite telemetry with open-loop Doppler correction.

Like most modern radios, the FCD uses fractional-N frequency synthesis with inherent restrictions on the actual set of frequencies that can be produced. Although its programming interface accepts frequencies in 1 Hz steps, the actual tuning step size varies with frequency; in the HF and VHF range it is 1000/2048 = 0.48828 Hz so it cannot produce exact 1 Hz frequency steps.

The 'radio' program works around this in two independent ways. First,

the hardware synthesizer programming formulas and constants from the FCD firmware are used to determine the actual LO frequency so any discrepancy can be corrected by the second LO. Second, only frequencies that the synthesizer can exactly produce are selected, restricting the synthesizer to a relative large step size. The step size varies with frequency so the worst (largest) value of 125 Hz is always used, with the difference again absorbed by the second LO. Although these two fixes are specific to the FCD, other SDRs undoubtedly have the same problem subject to the same workarounds -- if I can get the necessary details.

Multicasting and WiFi

High speed multicast streams can cause problems with many consumer grade WiFi base stations. Although multicasting is widely used for resource discovery (especially by Apple devices), the low data rates do not cause any problems. But a raw I/Q multicast stream from a FCD is about 6.5 Mb/s. This will cause many consumer-grade WiFi access points to roll over and die.

WiFi (especially the later versions) dynamically adapts to varying link conditions with a very wide range of transmission speeds, from only 1 Mb/s in 802.11b to over 1 Gb/s for 802.11ac. This is great for unicast traffic because the base station transmits to each client station at the fastest rate it can receive. If a client fails to acknowledge a transmission, the base station can retry at lower rates until it gets through.

The problem is that WiFi multicasts are not acknowledged, so to ensure that they reach everyone they are transmitted at a low and usually fixed data rate. Sometimes this fixed rate can be changed, but usually not. So a channel that typically operates at, say, 200 Mb/s may grind to a near halt when a lot of multicast traffic must be sent at only a few megabits/sec. If the multicast traffic arrives faster than it can be sent, the problem is obvious.

Several fixes and workarounds are available, but usually only in base stations designed for commercial environments. I have a pair of Engenius EAP600 access points at home that implement both solutions. This particular model is now discontinued but many current models from different manufacturers continue to provide them.

The first fix is IGMP Snooping. By listening in on the Internet Group Management Protocol (IGMP) messages sent by each computer to indicate the multicast groups it subscribes to, an Ethernet switch can avoid sending multicast traffic to ports where nobody wants it. A WiFi base station usually contains an internal Ethernet switch, and if it implements IGMP snooping it can avoid relaying multicast traffic to the radio channel when no one wants it.

But what if a wireless client wants the multicast traffic? The fix here is "multicast to unicast conversion". The base station keeps track (with IGMP snooping) of which wireless clients want traffic to which multicast groups, and each group member is sent its own copy of the multicast packet as a unicast at whatever speed is necessary to reach that client, which then acknowledges it like any other unicast packet. This works very well when only a few wireless clients subscribe to multicast groups and/or the links to those clients can operate at high speed, as is usually the case. The time required to

send each client its own copy as a high speed unicast is still far# ka9q-radio

Overview

Ka9q-radio is a collection of software defined radio (SDR) modules connected by the Internet Real Time Protocol (RTP) and IP multicasting. It is intended to facilitate experimentation and education, and to be easy to interface to data decoders and digital communication programs. It can also smoothly track satellite Doppler shift in an open-loop mode when fed velocity and acceleration information from an external program.

The ka9q-radio modules execute from the command line on either Linux or OSX. As yet there is no fancy graphical user interface. (This may come later, especially if someone volunteers to create one).

License
The License for this software is GPL3, as detailed [here](LICENSE).

Installation
Installation instructions are [here](INSTALLING.md).

Multicasting and the Real Time Protocol (RTP)

The ka9q-radio modules are designed for maximum versatility. Unlike shell pipelines or TCP/IP connections, ka9q-radio modules can be individually started and stopped at any time, and one can feed any number of others without prearrangement. They can run on the same computer or on different computers on a multicast-capable network (e.g., an Ethernet LAN).

RTP was originally designed for multicast use, though it is now used mainly in a unicast (point-to-point) mode for voice over IP (VoIP). It provides sequenced delivery, but like all real-time and multicast protocols it cannot use retransmissions to guarantee delivery. This distinguishes it from the Transmission Control Protocol (TCP) that can only be used in a point-to-point mode for data that can tolerate latency.

Ka9q-radio uses four RTP payload types: raw I/Q data with a custom receiver status header; mono or stereo 16-bit uncompressed PCM audio sampled at 48 kHz; and the new Opus codec.

It should be easy to add PCM/RTP input to any ham SDR program that uses a computer sound card to acquire receiver audio (e.g. WSJT-X, fldigi and many others). This would allow these programs to accept receive audio directly over the network from the ka9q 'radio' program, completely bypassing the computer's sound system and freeing it for other uses. This would be a much cleaner and more reliable interface as there would be no analog audio cables (real or "virtual"), "rigblaster" adapter boxes, audio ground loops, equalization problems, or tricky level adjustments! With multicasting, any number of programs can simultaneously process the same receiver output.

Parts of the ka9q-radio package are well suited to "turnkey" networked receiver applications such as receive-only APRS-to-Internet gateways and Broadcastify feeds. Service descriptions are provided for Linux udev and systemd to load the daemons at boot time.

Overview
Opus Compression and Audio Monitoring

Although PCM is best for digital decoders, it requires 800 kb/s for mono and 1.6 Mb/s for stereo. Much lower data rates suffice for human listening over WiFi or a remote Internet connection. I use the new Opus codec standardized by the Internet Engineering Task Force (IETF). An excellent open-source implementation is available, and it can be freely used without patent restrictions (unlike, e.g., ABME or AAC). Opus automatically switches algorithms as needed to do the best it can with whatever data rate it is given, from very high fidelity stereo audio at up to (an unnecessarily high) 510 kb/s down to monaural voice at only 6 kb/s. (VK5DGR's CODEC2 can handle even lower data rates, which is on my to-do list.) I typically use Opus at 32 kb/s, which is actually much higher than needed for communications-quality audio.

Popular media players like VLC support PCM and Opus so you can listen to ka9q-radio audio on many smartphones and tablets. Currently the player must receive audio multicasts, which usually means it must be on the same LAN as the computer(s) generating them. Remote unicast and tunneled multicast connectivity is on the to-do list.

The ka9q-radio package also includes 'monitor', a RTP audio player that can receive and mix several multicast audio streams to the local sound output. It has a novel 'pan' feature that uses gain and delay adjustments (<1 ms) to place each source at a user-chosen point in a stereo image. This makes it easier to distinguish multiple sources, e.g., in a round table. The 'monitor' program currently supports only Opus and PCM, though CODEC2 is again on the list.

Direct, low-latency access to multicast data on remote networks requires some form of IP multicast routing or tunneling that is not (yet) provided in this package. The remote multicast data you are most likely to want is audio, so here's a simple workaround for remote listening:

```
ssh remotesys 'pcmcat -2 pcm.vhf.mcast.local | opusenc --quiet --raw --bitrate 32 -
-' | play -q -
```

This remotely executes the 'pcmcat' command, which picks up the specified multicast group (which must be PCM audio with RTP types 10 or 11), forces the output to be stereo (which opusenc expects by default), and compresses it with the Opus codec to 32 kb/s. The compressed audio is then sent over the SSH channel to your local computer where the 'play' program decompresses and plays the audio. This requires the 'opus-tools' package on the remote system and the 'sox' package on the local system. Latency can be several seconds because of shell pipeline and TCP buffering.

There's no performance penalty to running Opus in stereo mode on mono data, so for simplicity the ka9q-radio package always uses Opus in this way.

The KA9Q 'radio' program

The heart of the ka9q-radio package is the program 'radio', an interactive general coverage receiver. It reads raw I/Q data multicast by a SDR front end, sending it tuning and gain adjustments as

needed. Radio's output is a mono or stereo PCM audio multicast stream.

If audio compression is needed, a separate 'opus' module transcodes PCM to Opus. I.e., it receives PCM, compresses it with Opus and multicasts it to a different IP address. This lets listeners and players select compressed or uncompressed audio by merely joining the corresponding multicast group.

The user interface is simple but powerful. It runs in text mode, which makes it easy and fast to run remotely. If desired, the interactive interface can be disabled entirely and all parameters given instead on the command line (e.g., in shell scripts).

Architecture

The 'radio' program accepts a generic I/Q (complex) sample stream multicast by a computer with a direct conversion ("zero IF") SDR front end. Many inexpensive SDR front ends are available, though at present only the AMSAT UK FUNcube Pro+ dongle (heretofore called the "FCD") is supported (with the 'funcube' module). The (mostly higher) sample rates produced by other receivers such as the SDRPlay and RTL-SDR are supported but only the FCD's 192 kHz sampling rate has actually been tested.

An I/Q stream at 192 kHz with 16 bit samples is a constant 6.5 Mb/s network load; this is no problem for modern Ethernet but it should be kept away from inexpensive WiFi base stations and public Internets. (More on problems with multicast and WiFi in the end notes).

Hardware Artifact Removal

Although most SDR front ends use direct conversion, the complete SDR front end/ka9q receiver combination actually forms a dual-conversion superhet: the first LO and mixer are in analog hardware and the second LO and mixer are in software. The first mixer, a quadrature (I/Q) type, produces a complex first IF centered on zero Hz and extending from -Fs/2 to +Fs/2, where Fs is the A/D sample rate. For the FCD, Fs = 192 kHz so the IF represents a "window" on the radio spectrum from 96 kHz below the first LO frequency to 96 kHz above. (The Nyquist or minimum sample rate for a complex signal is equal to the bandwidth; for a more familiar real signal, the Nyquist rate is twice the signal bandwidth.)

This is often called a "zero IF" architecture but is more accurately called a "low frequency IF". The actual first IF can be anywhere between -Fs/2 and +Fs/2 (including zero) provided it contains the entire signal bandwidth. The 10 kHz near each edge from a FCD is best avoided because of imperfect anti-alias filtering before the A/D converters, and I presume other SDR front ends have similar limitations.

All analog direct conversion SDR front ends produce a "DC spike" at 0 Hz from crosstalk from the first LO to the receiver input. On the FCD this spike is about -50 dBFS (decibels below full scale, i.e., A/D saturation). The first processing step in 'radio' completely removes it, but reciprocal mixing produces a "mound" of low frequency phase noise that cannot be removed. With the FCD this noise is only noticeable within a few kilohertz of zero and then only when the antenna is disconnected to remove external noise. But it is still easy

to avoid by simply shifting the signal to a higher IF. It is common in these designs to place the IF near one quarter of the sample rate, e.g., the range from +48 kHz to +51 kHz (or -51 kHz to -48 kHz) might be selected for 3 kHz wide SSB. The 'radio' program tries to keep the IF in one of these two ranges as the user changes frequency.

48 kHz is still an extremely low first IF by usual standards, as ordinary pre-mixer filters cannot reject an image only 96 kHz away. This problem is avoided by the use of complex I/Q signals that permit the frequency image to be canceled with digital signal processing. Good image rejection requires the 'radio' program to compensate for any slight gain and phase imbalances between the I and Q channels, and this is its second processing step. Typical values for the FCD are 0.07 dB of gain imbalance and 0.4 degrees of phase error, i.e., deviation from exactly 90 degrees between the two channels.

Hardware artifact removal was recently added to the 'funcube' and 'hackrf' modules. It is still present in 'radio' but is disabled by default.

Frequency conversion

The third step is to convert the first IF signal to baseband, i.e., a carrier frequency of 0 Hz. This is performed by the second (software) LO and mixer using complex arithmetic to suppress the unwanted image. The LO and mixer each require only one complex multiply per sample. No lookup tables are used, and sine and cosine calls are needed only when changing the LO frequency.

Another LO and mixer are optionally used for open-loop correction of satellite Doppler shift. The 'radio' program reads velocity and acceleration from an external program, calculates and applies the appropriate frequency offset and rate.

When Doppler steering is active, a second LO is added to sweep the frequency of the first, requiring only two more complex multiplies per sample. Sine and cosine calls are needed only when the frequency sweep rate is changed. The Doppler steering is so smooth and accurate that I have been able to copy CW from a LEO cubesat transmitting on 70cm. Even on a near-overhead path, the signal was rock stable, staying in a narrow CW filter. (Of course, this requires accurate orbital elements and current clock time. It might be possible to use observed frequency errors to correct a set of orbital elements.)

Filtering and Demodulation

The baseband signal is filtered with fast correlation. With complex signals, the passband can be asymmetric, e.g., +100 to +3000 Hz selects upper sideband (USB) and -100 to -3000 Hz selects LSB. A post-detection frequency shift is provided for CW; this is equivalent to re-tuning the radio while simultaneously sliding the filter to keep the signal at the same point in the passband. Modes other than SSB typically use filters centered on 0 Hz but this is not required; the edges can be individually varied, e.g. to suppress adjacent channel interference.

The filtered baseband signal is then fed to one of three demodulators: 'AM', 'FM' and 'Linear.'

AM Demodulator

The AM demodulator is an ordinary non-coherent envelope detector that simply measures the amplitude of the complex baseband signal, ignoring the phase. A squelch needs to be added to make it usable for aviation communications.

FM Demodulator

The FM demodulator handles narrow-band frequency or phase modulation by taking the phase of a baseband signal with the arctangent function and then differentiating it (frequency is the derivative of phase). The arctangent instruction is surprisingly fast even on a Raspberry Pi, so a lot of literature on fast arctangent approximations seems to be unnecessary.

Two submodes are provided. 'FM' has a high pass cut on the received audio below 300 Hz to remove PL/CTCSS tones and a standard -6 dB/octave audio de-emphasis above 300 Hz. This seems to be common amateur NBFM practice.

The 'FMF' (FM Flat) mode is straight FM without post-detection filtering so the signal is treble-heavy and any PL tones are audible. Both modes use an FFT to measure PL/CTCSS tone frequencies to 0.1 Hz accuracy. SNR, frequency offset and peak deviation are also measured and displayed.

A classic FM demodulator ignores signal amplitude, but I use it in an experimental threshold extension scheme. Complex baseband samples with amplitudes falling below a certain fraction of the average signal strength are blanked. (The amplitude threshold is empirically set at 0.55 of the average). This is very effective at removing "popcorn" noise as the SNR decreases to threshold. I'd like to see how it compares to standard threshold extension techniques such as a narrow PLL.

The FM squelch works from first principles: the signal-to-noise ratio is estimated and the squelch opens when it exce### **aprsfeed**eds +6 dB. This works so well that I've felt no need for a squelch adjustment. The squelch originally closed so quickly that there was no tail, but I added a short one to ensure that the tail end of a packet transmission isn't cut off.

After the squelch closes the silent PCM output (containing all 0's) continues for a few packets to flush any digital filters decoding the stream. The PCM audio stream then stops to reduce network load. The Opus transcoder, if used, also stops producing compressed packets. When the squelch reopens, the RTP marker bit is set to flag a jump in the sample count. The RTP timestamp indicates how many silent samples were suppressed. This allows programs using the input stream for timing to maintain a proper sample count.

Linear Demodulation

The 'linear' detector is used for all other modes, including SSB/CW, ISB (independent sideband) and raw I/Q. In linear mode the filter output is the audio output: both the I and Q channels for stereo modes and just the I channel for mono. Stereo is implied by I/Q and ISB. It

is optional for CW/SSB, where it produces an interesting effect on headphones because of the 90 degree phase shift (Hilbert transform) between the left (I) and right (Q) channels -- the sound no longer seems to come from a point in the middle of your brain. I think this might be useful in decreasing fatigue during long contests.

The ISB mode is the same as raw I/Q except that the filter cross-adds the positive and negative spectral components to put the lower sideband on the I (left) channel and the upper sideband on Q (right).

A PLL can coherently track an AM carrier for synchronous detection, which is usually much less noisy than regular AM envelope detection. The carrier frequency offset is averaged and displayed; This is useful for calibration of the SDR TCXO against an external reference such as WWV/WWVH or the pilot carrier of a local ATSC digital TV transmitter.

A squarer can be enabled so the PLL can track the suppressed carrier in DSB-AM and BPSK.

User Interface

The 'radio' program has a textual user interface that uses the "ncurses" library. It looks primitive by 2018 standards, but it is fully functional and very efficient over slow Internet paths. The `h` key pops up a summary of commands. The textual user interface can also be completely disabled and all receiver parameters given on the command line, e.g, for invocation from a shell script in a dedicated system.

Startup parameters are stored as files in the .radiostate directory in the user's home directory. The parameters include the input and output multicast domain names (or IP addresses), the tuning frequency and step size, the filter settings and detector mode. The correlator block size and FIR filter length can also be set here; note they cannot be changed after the program starts except by stopping and restarting it.

Invoking 'radio' with the name of a startup file loads its settings. The state of the radio can be saved to a state file with the `w` key. On normal termination the radio state is also stored as "default". It will be automatically loaded on the next invocation if no explicit name is given.

The various operating modes ("AM", "USB", etc) are defined in /usr/local/share/ka9q-radio/modes.txt. Each entry specifies a demodulator along with filter parameters, the post-detection frequency shift, AGC responses and option flags. For example, "USB" selects the "linear" demodulator, a filter passband of +100 to +3000 Hz, a zero post-detection frequency offset, an AGC attack rate of -50 dB/sec, an AGC recovery rate of +6 dB/sec, an AGC hang time of 1.1 seconds and mono output.

The textual screen display consists of the following status windows:

The "Tuning" window shows the RF, LO and IF frequencies. All can be changed by the user. If Doppler correction is active, it is also shown here but cannot be modified by the user.

The user can change frequency with the arrow keys, a Griffin Power Mate USB knob, a mouse wheel, or by typing a complete frequency into the pop up window invoked with the `f` command. The cursor indicates the digit that will change with the USB knob, mouse wheel or up and down arrow keys. The cursor can be moved to a different digit with the left and right arrow keys, and it can be moved to the next field with the TAB key. (Shift-TAB moves to the previous field.)

The "Carrier" and "Center" fields usually differ only in the linear modes when the filter edges are not symmetric around 0 Hz. The "Carrier" field shows the (suppressed) carrier frequency that comes out at zero Hz in the receiver output while the "Center" field indicates the center of the receiver passband. For example, in USB mode with a +100 to +3000 Hz frequency response, the "Center" frequency will be (100 + 3000)/2 = 1550 Hz above the carrier frequency.

The "shift" setting shifts the entire audio output spectrum; the indicated carrier frequency appears at the selected frequency in the audio output rather than 0 Hz. This can be useful for CW operation but is not required.

The radio is most easily tuned through the "Carrier" or "Center" fields, but it can also be indirectly tuned through the First LO and IF fields. The displayed LO frequency includes the estimated TCXO frequency error and the effects of fractional-N frequency synthesis (see the end notes).

Any number of copies of 'radio' can process the same I/Q stream provided they share the same frequency range. When there is a conflict, the last radio to re-tune the SDR LO wins. The other copies will not try to fight it; without further user input they will simply wait until the LO is again within range. To avoid unintentional changes in the LO frequency, tab to the First LO field and hit the `l` key; this will underline the entry to indicate that the field is locked. The Carrier/Center frequency fields can also be locked.

The LO frequency in an I/Q recording is set at recording time and cannot be changed during playback.

The "Signal" window shows signal, noise and SNR estimates at various points in the signal path. Levels are in dBFS, decibels below full scale (A/D saturation) as there is no way to know the analog gain ahead of the A/D converters without careful calibration, which is invariably frequency dependent. These are all output-only fields.

The "Info" window shows available information on the name of the tuned channel or frequency band. For a ham band, the allowed emissions and required license class are given. These are all output-only fields, taken from /usr/local/share/ka9q-radio/bandplan.txt.

The "Filtering" window shows the pre-detection filter settings. The first four parameters are user-adjustable as part of the field sequence selected with th## **'radio' in quiet/daemon mode**e TAB key. Smaller Kaiser betas give sharper
filters but poorer stop-band attenuation; 0 is equivalent to rectangular windowing, i.e., no windowing at all. Larger betas give wider transitions but better ultimate rejection outside the passband.

The block size and FIR (which determine the FFT frequency bin size and filter delay) can only be set at startup from a state file or on the command line. [See the later discussion about sample rates and decimation.]

The "Demodulator" window is mode-specific and output-only. The AM and linear modes include a simple AGC and the relative audio gain is shown (it needs a real S-meter). The FM mode shows the frequency offset, peak deviation and PL/CTCSS tone (if present). When the selected mode and options permit the signal-to-noise ratio to be estimated, it is shown. (These estimates are done differently from the SNR estimate in the "Signal" window so they may not exactly match.)

The "Options" window selects the various options for the "Linear" demodulator. You can select an entry with the mouse or type the mode into a pop up box with the `o` command.

The output-only "SDR Hardware" window shows the nominal A/D sample rate, TCXO offset, nominal tuner frequency (uncorrected by the TCXO offset estimate), analog gain settings and gain and phase imbalance estimates.

The "Modes" window allows a predefined operating mode to be conveniently selected with the mouse.

The "I/O" window gives information on the input and output multicast network streams: IP addresses or host names; port numbers; packet, sample and error counts; and a SDR timestamp useful when playing back recorded I/Q data. It also estimates the true input A/D sample rate against the local computer time-of-day clock.

A "Debug" area at the very bottom of the screen shows version information and various test and debugging messages.

Textual user input entered through pop up menus; for example the `f` key pops up a box into which a frequency can be directly entered as `147m435` (147.435 MHz), `760k` (760 kHz) or `1g296296` (1296.296 MHz). If no letters are used to indicate the magnitude of the decimal point, a 'best guess' is made to produce a valid frequency; e.g. 14313 will give 14.313 MHz, not 14.313 kHz (which is below the FCD's range). Note that some ranges are inherently ambiguous; e.g., `500` could be either 500 kHz or 500 MHz so they should be typed with explicit decimal points.

'radio' in quiet/daemon mode

The 'radio' program can run in a totally 'quiet' mode with no user interface. This is suitable for automatic execution at boot time for fixed channel operation (e.g., receiving and reporting APRS transmissions). Two Linux systemd service descriptions are provided: 'radio34' and 'radio39'; the first creates an instance of 'radio' listening on 144.39 MHz in FM mode and the latter does the same on 144.39 MHz. Thise two instances can share the same Funcube dongle because their frequencies are only 50 kHz apart, within the 192 kHz bandwidth of the Funcube dongle.

144.39 MHz is the standard APRS frequency for North America; 144.34 MHz is the secondary APRS frequency used by WB8ELK's 15-gram 'pico tracker' for long duration balloon flights.

aprsfeed## Other ka9q-radio Modules

Other modules in the package provide miscellaneous functions and/or start more complex planned features. None have especially complex user interfaces; most take only a few command line arguments and can run as background daemons. They are given in alphabetical order

aprs

This unfinished module accepts decoded AX.25 frames from the 'packet' module, extracts APRS position data from a selected station, computes the azimuth, elevation and range to that station from a specified point, and commands antenna rotors to point at the transmitting station. This was written specifically for tracking high altitude balloons.

aprsfeed

This module also accepts decoded AX.25 frames from the 'packet' module. It feeds those frames to the international APRS network so position reports will automatically appear on sites such as www.aprs.fi. It can be run as a background daemon.

funcube

This module takes the digital IF from a locally connected FCD and multicasts it over the local LAN. It accepts unicast commands to tune the radio and set analog gains. It will automatically reduce gain to avoid A/D saturation. Similar modules will be written for other SDR front ends such as the SDRPlay and RTL-SDR that will either talk directly to the hardware or through "shimware" such as SoapySDR.

The 'funcube' module can be automatically loaded at boot time (or at device insertion) as a background daemon by the Linux udev/systemd subsystem. A udev rule file and a systemd service file are provided. Note that the latter specifies the multicast group for the digital IF data, and this will probably require local configuration.

In daemon mode, 'funcube' writes its status to /run/funcube#/status, where '#' is the device number (starting at 0). Scripts for two devices are provided. The Raspberry Pi can support two devices, but I prefer one Pi per dongle when possible.

hackrf

This module is functionally equivalent to'funcube' except for the HackRF One. Only reception is currently supported. The HackRF supports sample rates up to 20 MHz but has only a 8-bit A/D, so by default it samples at a high speed and decimates to 192 kHz, the same as the funcube. The USB data rate and decimation processing load is too great for a Raspberry Pi, but it will run on a low-end x86 system.

Like 'funcube', 'hackrf' can be automatically loaded at boot time under Linux.

In daemon mode, 'hackrf' writes its status to /run/hackrf0/status. (Only one hackrf device is currently supported, but this can be changed.)

iqrecord and iqplay

These modules perform the functions suggested by their names. 'iqrecord' accepts multicast I/Q or PCM audio streams and writes them to disk. Gaps in the multicast stream due to lost packets or silence suppression are seeked over in a recording to maintain the correct sample count and playback timing. With a file system that supports "holes", disk blocks need not be allocated to these silent periods.

Raw I/Q streams are written into files named as 'iqrecord-xxxxxx-n' where xxxxxx is the RTP SSRC (Stream Source Identifier) and 'n' is a number incremented to avoid overwriting existing recordings. PCM streams are written as 'pcmrecord-xxxxxx-n'. Both files are written as raw, header-less 16-bit PCM with all meta information in extended file attributes.

Most but not all modern file systems support extended attributes; a notable exception is Microsoft's VFAT32 often used as a least-common-denominator for file exchange between different operating systems. Warning! Many file copy and archiving utilities do not preserve extended attributes, at least not by default.

I/Q and PCM recordings have many uses. An I/Q recording preserves everything within the front end passband regardless of modulation type or bandwidth (provided it fits within the receiver passband). This is useful for testing and for demonstrating the 'radio' program when an antenna is not available.

I/Q recordings are especially useful as backups during critical events, such as a satellite pass or balloon flight. An I/Q recorder placed next to the SDR hardware depends only on the antenna, SDR hardware and I/Q multicast program (e.g. 'funcube'). This can guard against bugs (or the outright failure) of downstream network or processing elements, e.g., a digital data demodulator. After the demodulator is fixed, the recording can be played back to recover the data.

And of course nothing limits you to just one I/Q recording.

'iqrecord' is an excellent example of the versatility of multicasting; it can just sit quietly in the corner backing everything up without impairing any other element that might be processing the same stream.

The 'iqplay' module plays back I/Q recordings (only -- no PCM support at present) using the meta data contained in the external file attributes. It can also read a raw I/Q sample stream from standard input to simulate SDR front end hardware.

modulate

A simple (and unfinished) test modulator that takes baseband audio, amplitude modulates it on a specified carrier frequency, and emits it on standard output as an I/Q sample stream.

opus

The 'opus' module was described earlier as an optional "transcoder" that accepts uncompressed PCM (either mono or stereo) and produces a

lower bit rate compressed version with the Opus codec. Options include a range of block sizes from 2.5 milliseconds to 120 ms. The larger blocks are useful only to reduce packet rate and header overhead; they provide no additional compression and are supported only by more recent versions of the Opus library. The default is 20 ms, which is long enough to give good compression but short enough to be essentially unnoticeable.

The compressed output data rate is specified by a command line parameter that defaults to 32 kb/s. This produces surprisingly good audio even on stereo music.

The Opus codec also supports a VOX-like "discontinuous" mode well suited to half-duplex push-to-talk amateur operation. When silence is detected, it automatically drops to sending "comfort noise" at only 3 packets per second even as the PCM input stream continues. If the PCM input stream stops, the Opus output stops regardless of the discontinuous setting.

Opus streams are always stereo even when the audio is mono. There is no capacity penalty and it simplifies things.

'Opus' can be run as a daemon; systemd 'service' config files are provided.

opussend

This is a standalone utility that takes PCM audio from a local sound interface, compresses it with Opus and multicasts it as an RTP network stream.

packet

This module is my first digital demodulator module for the ka9q-radio package. It accepts PCM audio from the 'radio' program, demodulates the ancient Bell 202A AFSK modem tones and decodes AX.25 frames. The decoded frames are multicast as RTP/UDP packets. The decoded frames can also optionally be displayed on the console. Otherwise the module can run as an unattended daemon.

'Packet' can be run as a daemon; a systemd 'service' file is provided.

pcmsend

This is the same as opussend, except that the output stream is uncompressed 48 kHz PCM (mono or stereo).

pcmcat

This joins a specified multicast group, which must carry uncompressed PCM audio (RTP types 10 or 11) and emits the PCM stream on standard output. This is useful for piping into an audio compressor for remote transmission.

Footnotes and Side bars

Sample Rates and Decimation

The I/Q input sample rate must be an integer multiple of the 48 kHz audio output rate. Decimation is performed as a byproduct of the fast

correlator used for pre-detection filtering. Since the input FFT in a fast correlator executes at the input sample rate, CPU loading will increase. Although the CPU loading at 192 kHz is small even on a Raspberry Pi, faster and simpler decimators are in the works to support much higher sampling rates.

By default the filter decimates the FCD 192 kHz A/D input sample rate by a factor of 4 to a 48 kHz audio output. Other SDR front ends with higher sample rates will require correspondingly higher decimation ratios. Although 48 kHz may seem excessive for communications-grade audio, I don't recommend reducing it. 48 kHz is supported by nearly every audio D/A, and it still uses only 0.154% of a gigabit Ethernet link.# ka9q-radio

Overview

Ka9q-radio is a collection of software defined radio (SDR) modules connected by the Internet Real Time Protocol (RTP) and IP multicasting. It is intended to facilitate experimentation and education, and to be easy to interface to data decoders and digital communication programs. It can also smoothly track satellite Doppler shift in an open-loop mode when fed velocity and acceleration information from an external program.

The ka9q-radio modules execute from the command line on either Linux or OSX. As yet there is no fancy graphical user interface. (This may come later, especially if someone volunteers to create one).

License
The License for this software is GPL3, as detailed [here](LICENSE).

Installation
Installation instructions are [here](INSTALLING.md).

Multicasting and the Real Time Protocol (RTP)

The ka9q-radio modules are designed for maximum versatility. Unlike shell pipelines or TCP/IP connections, ka9q-radio modules can be individually started and stopped at any time, and one can feed any number of others without prearrangement. They can run on the same computer or on different computers on a multicast-capable network (e.g., an Ethernet LAN).

RTP was originally designed for multicast use, though it is now used mainly in a unicast (point-to-point) mode for voice over IP (VoIP). It provides sequenced delivery, but like all real-time and multicast protocols it cannot use retransmissions to guarantee delivery. This distinguishes it from the Transmission Control Protocol (TCP) that can only be used in a point-to-point mode for data that can tolerate latency.

Ka9q-radio uses four RTP payload types: raw I/Q data with a custom receiver status header; mono or stereo 16-bit uncompressed PCM audio sampled at 48 kHz; and the new Opus codec.

It should be easy to add PCM/RTP input to any ham SDR program that uses a computer sound card to acquire receiver audio (e.g. WSJT-X, fldigi and many others). This would allow these programs to accept

receive audio directly over the network from the ka9q 'radio' program, completely bypassing the computer's sound system and freeing it for other uses. This would be a much cleaner and more reliable interface as there would be no analog audio cables (real or "virtual"), "rigblaster" adapter boxes, audio ground loops, equalization problems, or tricky level adjustments! With multicasting, any number of programs can simultaneously process the same receiver output.

Parts of the ka9q-radio package are well suited to "turnkey" networked receiver applications such as receive-only APRS-to-Internet gateways and Broadcastify feeds. Service descriptions are provided for Linux udev and systemd to load the daemons at boot time.

Opus Compression and Audio Monitoring

Although PCM is best for digital decoders, it requires 800 kb/s for mono and 1.6 Mb/s for stereo. Much lower data rates suffice for human listening over WiFi or a remote Internet connection. I use the new Opus codec standardized by the Internet Engineering Task Force (IETF). An excellent open-source implementation is available, and it can be freely used without patent restrictions (unlike, e.g., ABME or AAC). Opus automatically switches algorithms as needed to do the best it can with whatever data rate it is given, from very high fidelity stereo audio at up to (an unnecessarily high) 510 kb/s down to monaural voice at only 6 kb/s. (VK5DGR's CODEC2 can handle even lower data rates, which is on my to-do list.) I typically use Opus at 32 kb/s, which is actually much higher than needed for communications-quality audio.

Popular media players like VLC support PCM and Opus so you can listen to ka9q-radio audio on many smartphones and tablets. Currently the player must receive audio multicasts, which usually means it must be on the same LAN as the computer(s) generating them. Remote unicast and tunneled multicast connectivity is on the to-do list.

The ka9q-radio package also includes 'monitor', a RTP audio player that can receive and mix several multicast audio streams to the local sound output. It has a novel 'pan' feature that uses gain and delay adjustments (<1 ms) to place each source at a user-chosen point in a stereo image. This makes it easier to distinguish multiple sources, e.g., in a round table. The 'monitor' program currently supports only Opus and PCM, though CODEC2 is again on the list.

Direct, low-latency access to multicast data on remote networks requires some form of IP multicast routing or tunneling that is not (yet) provided in this package. The remote multicast data you are most likely to want is audio, so here's a simple workaround for remote listening:

```
ssh remotesys 'pcmcat -2 pcm.vhf.mcast.local | opusenc --quiet --raw --bitrate 32 -
-' | play -q -
```

This remotely executes the 'pcmcat' command, which picks up the specified multicast group (which must be PCM audio with RTP types 10 or 11), forces the output to be stereo (which opusenc expects by default), and compresses it with the Opus codec to 32 kb/s. The compressed audio is then sent over the SSH channel to your local computer where the 'play' program decompresses and plays the audio. This requires the 'opus-tools' package on the remote system and the 'sox'

package on the local system. Latency can be several seconds because of shell pipeline and TCP buffering.

There's no performance penalty to running Opus in stereo mode on mono data, so for simplicity the ka9q-radio package always uses Opus in this way.

The KA9Q 'radio' program

The heart of the ka9q-radio package is the program 'radio', an interactive general coverage receiver. It reads raw I/Q data multicast by a SDR front end, sending it tuning and gain adjustments as needed. Radio's output is a mono or stereo PCM audio multicast stream.

If audio compression is needed, a separate 'opus' module transcodes PCM to Opus. I.e., it receives PCM, compresses it with Opus and multicasts it to a different IP address. This lets listeners and players select compressed or uncompressed audio by merely joining the corresponding multicast group.

The user interface is simple but powerful. It runs in text mode, which makes it easy and fast to run remotely. If desired, the interactive interface can be disabled entirely and all parameters given instead on the command line (e.g., in shell scripts).

Architecture

The 'radio' program accepts a generic I/Q (complex) sample stream multicast by a computer with a direct conversion ("zero IF") SDR front end. Many inexpensive SDR front ends are available, though at present only the AMSAT UK FUNcube Pro+ dongle (heretofore called the "FCD") is supported (with the 'funcube' module). The (mostly higher) sample rates produced by other receivers such as the SDRPlay and RTL-SDR are supported but only the FCD's 192 kHz sampling rate has actually been tested.

An I/Q stream at 192 kHz with 16 bit samples is a constant 6.5 Mb/s network load; this is no problem for modern Ethernet but it should be kept away from inexpensive WiFi base stations and public Internets. (More on problems with multicast and WiFi in the end notes).

Hardware Artifact Removal

Although most SDR front ends use direct conversion, the complete SDR front end/ka9q receiver combination actually forms a dual-conversion superhet: the first LO and mixer are in analog hardware and the second LO and mixer are in software. The first mixer, a quadrature (I/Q) type, produces a complex first IF centered on zero Hz and extending from -Fs/2 to +Fs/2, where Fs is the A/D sample rate. For the FCD, Fs = 192 kHz so the IF represents a "window" on the radio spectrum from 96 kHz below the first LO frequency to 96 kHz above. (The Nyquist or minimum sample rate for a complex signal is equal to the bandwidth; for a more familiar real signal, the Nyquist rate is twice the signal bandwidth.)

This is often called a "zero IF" architecture but is more accurately called a "low frequency IF". The actual first IF can be anywhere between -Fs/2 and +Fs/2 (including zero) provided it contains the entire signal bandwidth. The 10 kHz near each edge from a FCD

is best avoided because of imperfect anti-alias filtering before the A/D converters, and I presume other SDR front ends have similar limitations.

All analog direct conversion SDR front ends produce a "DC spike" at 0 Hz from crosstalk from the first LO to the receiver input. On the FCD this spike is about -50 dBFS (decibels below full scale, i.e., A/D saturation). The first processing step in 'radio' completely removes it, but reciprocal mixing produces a "mound" of low frequency phase noise that cannot be removed. With the FCD this noise is only noticeable within a few kilohertz of zero and then only when the antenna is disconnected to remove external noise. But it is still easy to avoid by simply shifting the signal to a higher IF. It is common in these designs to place the IF near one quarter of the sample rate, e.g., the range from +48 kHz to +51 kHz (or -51 kHz to -48 kHz) might be selected for 3 kHz wide SSB. The 'radio' program tries to keep the IF in one of these two ranges as the user changes frequency.

48 kHz is still an extremely low first IF by usual standards, as ordinary pre-mixer filters cannot reject an image only 96 kHz away. This problem is avoided by the use of complex I/Q signals that permit the frequency image to be canceled with digital signal processing. Good image rejection requires the 'radio' program to compensate for any slight gain and phase imbalances between the I and Q channels, and this is its second processing step. Typical values for the FCD are 0.07 dB of gain imbalance and 0.4 degrees of phase error, i.e., deviation from exactly 90 degrees between the two channels.

Hardware artifact removal was recently added to the 'funcube' and 'hackrf' modules. It is still present in 'radio' but is disabled by default.

Frequency conversion

The third step is to convert the first IF signal to baseband, i.e., a carrier frequency of 0 Hz. This is performed by the second (software) LO and mixer using complex arithmetic to suppress the unwanted image. The LO and mixer each require only one complex multiply per sample. No lookup tables are used, and sine and cosine calls are needed only when changing the LO frequency.

Another LO and mixer are optionally used for open-loop correction of satellite Doppler shift. The 'radio' program reads velocity and acceleration from an external program, calculates and applies the appropriate frequency offset and rate.

When Doppler steering is active, a second LO is added to sweep the frequency of the first, requiring only two more complex multiplies per sample. Sine and cosine calls are needed only when the frequency sweep rate is changed. The Doppler steering is so smooth and accurate that I have been able to copy CW from a LEO cubesat transmitting on 70cm. Even on a near-overhead path, the signal was rock stable, staying in a narrow CW filter. (Of course, this requires accurate orbital elements and current clock time. It might be possible to use observed frequency errors to correct a set of orbital elements.)

Filtering and Demodulation

The baseband signal is filtered with fast correlation. With complex signals, the passband can be asymmetric, e.g., +100 to +3000 Hz selects upper sideband (USB) and -100 to -3000 Hz selects LSB. A post-detection frequency shift is provided for CW; this is equivalent to re-tuning the radio while simultaneously sliding the filter to keep the signal at the same point in the passband. Modes other than SSB typically use filters centered on 0 Hz but this is not required; the edges can be individually varied, e.g. to suppress adjacent channel interference.

The filtered baseband signal is then fed to one of three demodulators: 'AM', 'FM' and 'Linear.'

AM Demodulator

The AM demodulator is an ordinary non-coherent envelope detector that simply measures the amplitude of the complex baseband signal, ignoring the phase. A squelch needs to be added to make it usable for aviation communications.

FM Demodulator

The FM demodulator handles narrow-band frequency or phase modulation by taking the phase of a baseband signal with the arctangent function and then differentiating it (frequency is the derivative of phase). The arctangent instruction is surprisingly fast even on a Raspberry Pi, so a lot of literature on fast arctangent approximations seems to be unnecessary.

Two submodes are provided. 'FM' has a high pass cut on the received audio below 300 Hz to remove PL/CTCSS tones and a standard -6 dB/octave audio de-emphasis above 300 Hz. This seems to be common amateur NBFM practice.

The 'FMF' (FM Flat) mode is straight FM without post-detection filtering so the signal is treble-heavy and any PL tones are audible. Both modes use an FFT to measure PL/CTCSS tone frequencies to 0.1 Hz accuracy. SNR, frequency offset and peak deviation are also measured and displayed.

A classic FM demodulator ignores signal amplitude, but I use it in an experimental threshold extension scheme. Complex baseband samples with amplitudes falling below a certain fraction of the average signal strength are blanked. (The amplitude threshold is empirically set at 0.55 of the average). This is very effective at removing "popcorn" noise as the SNR decreases to threshold. I'd like to see how it compares to standard threshold extension techniques such as a narrow PLL.

The FM squelch works from first principles: the signal-to-noise ratio is estimated and the squelch opens when it exceeds +6 dB. This works so well that I've felt no need for a squelch adjustment. The squelch originally closed so quickly that there was no tail, but I added a short one to ensure that the tail end of a packet transmission isn't cut off.

After the squelch closes the silent PCM output (containing all 0's) continues for a few packets to flush any digital filters decoding the stream. The PCM audio stream then stops to reduce network load. The

Opus transcoder, if used, also stops producing compressed packets. When the squelch reopens, the RTP marker bit is set to flag a jump in the sample count. The RTP timestamp indicates how many silent samples were suppressed. This allows programs using the input stream for timing to maintain a proper sample count.

Linear Demodulation

The 'linear' detector is used for all other modes, including SSB/CW, ISB (independent sideband) and raw I/Q. In linear mode the filter output is the audio output: both the I and Q channels for stereo modes and just the I channel for mono. Stereo is implied by I/Q and ISB. It is optional for CW/SSB, where it produces an interesting effect on headphones because of the 90 degree phase shift (Hilbert transform) between the left (I) and right (Q) channels -- the sound no longer seems to come from a point in the middle of your brain. I think this might be useful in decreasing fatigue during long contests.

The ISB mode is the same as raw I/Q except that the filter cross-adds the positive and negative spectral components to put the lower sideband on the I (left) channel and the upper sideband on Q (right).

A PLL can coherently track an AM carrier for synchronous detection, which is usually much less noisy than regular AM envelope detection. The carrier frequency offset is averaged and displayed; This is useful for calibration of the SDR TCXO against an external reference such as WWV/WWVH or the pilot carrier of a local ATSC digital TV transmitter.

A squarer can be enabled so the PLL can track the suppressed carrier in DSB-AM and BPSK.

User Interface

The 'radio' program has a textual user interface that uses the "ncurses" library. It looks primitive by 2018 standards, but it is fully functional and very efficient over slow Internet paths. The `h` key pops up a summary of commands. The textual user interface can also be completely disabled and all receiver parameters given on the command line, e.g, for invocation from a shell script in a dedicated system.

Startup parameters are stored as files in the .radiostate directory in the user's home directory. The parameters include the input and output multicast domain names (or IP addresses), the tuning frequency and step size, the filter settings and detector mode. The correlator block size and FIR filter length can also be set here; note they cannot be changed after the program starts except by stopping and restarting it.

Invoking 'radio' with the name of a startup file loads its settings. The state of the radio can be saved to a state file with the `w` key. On normal termination the radio state is also stored as "default". It will be automatically loaded on the next invocation if no explicit name is given.

The various operating modes ("AM", "USB", etc) are defined in /usr/local/share/ka9q-radio/modes.txt. Each entry specifies a

demodulator along with filter parameters, the post-detection frequency shift, AGC responses and option flags. For example, "USB" selects the "linear" demodulator, a filter passband of +100 to +3000 Hz, a zero post-detection frequency offset, an AGC attack rate of -50 dB/sec, an AGC recovery rate of +6 dB/sec, an AGC hang time of 1.1 seconds and mono output.

The textual screen display consists of the following status windows:

The "Tuning" window shows the RF, LO and IF frequencies. All can be changed by the user. If Doppler correction is active, it is also shown here but cannot be modified by the user.

The user can change frequency with the arrow keys, a Griffin Power Mate USB knob, a mouse wheel, or by typing a complete frequency into the pop up window invoked with the `f` command. The cursor indicates the digit that will change with the USB knob, mouse wheel or up and down arrow keys. The cursor can be moved to a different digit with the left and right arrow keys, and it can be moved to the next field with the TAB key. (Shift-TAB moves to the previous field.)

The "Carrier" and "Center" fields usually differ only in the linear modes when the filter edges are not symmetric around 0 Hz. The "Carrier" field shows the (suppressed) carrier frequency that comes out at zero Hz in the receiver output while the "Center" field indicates the center of the receiver passband. For example, in USB mode with a +100 to +3000 Hz frequency response, the "Center" frequency will be (100 + 3000)/2 = 1550 Hz above the carrier frequency.

The "shift" setting shifts the entire audio output spectrum; the indicated carrier frequency appears at the selected frequency in the audio output rather than 0 Hz. This can be useful for CW operation but is not required.

The radio is most easily tuned through the "Carrier" or "Center" fields, but it can also be indirectly tuned through the First LO and IF fields. The displayed LO frequency includes the estimated TCXO frequency error and the effects of fractional-N frequency synthesis (see the end notes).

Any number of copies of 'radio' can process the same I/Q stream provided they share the same frequency range. When there is a conflict, the last radio to re-tune the SDR LO wins. The other copies will not try to fight it; without further user input they will simply wait until the LO is again within range. To avoid unintentional changes in the LO frequency, tab to the First LO field and hit the `l` key; this will underline the entry to indicate that the field is locked. The Carrier/Center frequency fields can also be locked.

The LO frequency in an I/Q recording is set at recording time and cannot be changed during playback.

The "Signal" window shows signal, noise and SNR estimates at various points in the signal path. Levels are in dBFS, decibels below full scale (A/D saturation) as there is no way to know the analog gain ahead of the A/D converters without careful calibration, which is invariably frequency dependent. These are all output-only fields.

The "Info" window shows available information on the name of the tuned channel or frequency band. For a ham band, the allowed emissions and required license class are given. These are all output-only fields, taken from /usr/local/share/ka9q-radio/bandplan.txt.

The "Filtering" window shows the pre-detection filter settings. The first four parameters are user-adjustable as part of the field sequence selected with the TAB key. Smaller Kaiser betas give sharper filters but poorer stop-band attenuation; 0 is equivalent to rectangular windowing, i.e., no windowing at all. Larger betas give wider transitions but better ultimate rejection outside the passband. The block size and FIR (which determine the FFT frequency bin size and filter delay) can only be set at startup from a state file or on the command line. [See the later discussion about sample rates and decimation.]

The "Demodulator" window is mode-specific and output-only. The AM and linear modes include a simple AGC and the relative audio gain is shown (it needs a real S-meter). The FM mode shows the frequency offset, peak deviation and PL/CTCSS tone (if present). When the selected mode and options permit the signal-to-noise ratio to be estimated, it is shown. (These estimates are done differently from the SNR estimate in the "Signal" window so they may not exactly match.)

The "Options" window selects the various options for the "Linear" demodulator. You can select an entry with the mouse or type the mode into a pop up box with the `o` command.

The output-only "SDR Hardware" window shows the nominal A/D sample rate, TCXO offset, nominal tuner frequency (uncorrected by the TCXO offset estimate), analog gain settings and gain and phase imbalance estimates.

The "Modes" window allows a predefined operating mode to be conveniently selected with the mouse.

The "I/O" window gives information on the input and output multicast network streams: IP addresses or host names; port numbers; packet, sample and error counts; and a SDR timestamp useful when playing back recorded I/Q data. It also estimates the true input A/D sample rate against the local computer time-of-day clock.

A "Debug" area at the very bottom of the screen shows version information and various test and debugging messages.

Textual user input entered through pop up menus; for example the `f` key pops up a box into which a frequency can be directly entered as `147m435` (147.435 MHz), `760k` (760 kHz) or `1g296296` (1296.296 MHz). If no letters are used to indicate the magnitude of the decimal point, a 'best guess' is made to produce a valid frequency; e.g. 14313 will give 14.313 MHz, not 14.313 kHz (which is below the FCD's range). Note that some ranges are inherently ambiguous; e.g., `500` could be either 500 kHz or 500 MHz so they should be typed with explicit decimal points.

'radio' in quiet/daemon mode

The 'radio' program can run in a totally 'quiet' mode with no user interface. This is suitable for automatic execution at boot time for

fixed channel operation (e.g., receiving and reporting APRS transmissions). Two Linux systemd service descriptions are provided: 'radio34' and 'radio39'; the first creates an instance of 'radio' listening on 144.39 MHz in FM mode and the latter does the same on 144.39 MHz. Thise two instances can share the same Funcube dongle because their frequencies are only 50 kHz apart, within the 192 kHz bandwidth of the Funcube dongle.

144.39 MHz is the standard APRS frequency for North America; 144.34 MHz is the secondary APRS frequency used by WB8ELK's 15-gram 'pico tracker' for long duration balloon flights.

Other ka9q-radio Modules

Other modules in the package provide miscellaneous functions and/or start more complex planned features. None have especially complex user interfaces; most take only a few command line arguments and can run as background daemons. They are given in alphabetical order

aprs

This unfinished module accepts decoded AX.25 frames from the 'packet' module, extracts APRS position data from a selected station, computes the azimuth, elevation and range to that station from a specified point, and commands antenna rotors to point at the transmitting station. This was written specifically for tracking high altitude balloons.

aprsfeed

This module also accepts decoded AX.25 frames from the 'packet' module. It feeds those frames to the international APRS network so position reports will automatically appear on sites such as www.aprs.fi. It can be run as a background daemon.

funcube

This module takes the digital IF from a locally connected FCD and multicasts it over the local LAN. It accepts unicast commands to tune the radio and set analog gains. It will automatically reduce gain to avoid A/D saturation. Similar modules will be written for other SDR front ends such as the SDRPlay and RTL-SDR that will either talk directly to the hardware or through "shimware" such as SoapySDR.

The 'funcube' module can be automatically loaded at boot time (or at device insertion) as a background daemon by the Linux udev/systemd subsystem. A udev rule file and a systemd service file are provided. Note that the latter specifies the multicast group for the digital IF data, and this will probably require local configuration.

In daemon mode, 'funcube' writes its status to /run/funcube#/status, where '#' is the device number (starting at 0). Scripts for two devices are provided. The Raspberry Pi can support two devices, but I prefer one Pi per dongle when possible.

hackrf

This module is functionally equivalent to'funcube' except for the HackRF One. Only reception is currently supported. The HackRF supports

sample rates up to 20 MHz but has only a 8-bit A/D, so by default it samples at a high speed and decimates to 192 kHz, the same as the funcube. The USB data rate and decimation processing load is too great for a Raspberry Pi, but it will run on a low-end x86 system.

Like 'funcube', 'hackrf' can be automatically loaded at boot time under Linux.

In daemon mode, 'hackrf' writes its status to /run/hackrf0/status. (Only one hackrf device is currently supported, but this can be changed.)

iqrecord and iqplay

These modules perform the functions suggested by their names. 'iqrecord' accepts multicast I/Q or PCM audio streams and writes them to disk. Gaps in the multicast stream due to lost packets or silence suppression are seeked over in a recording to maintain the correct sample count and playback timing. With a file system that supports "holes", disk blocks need not be allocated to these silent periods.

Raw I/Q streams are written into files named as 'iqrecord-xxxxxx-n' where xxxxxx is the RTP SSRC (Stream Source Identifier) and 'n' is a number incremented to avoid overwriting existing recordings. PCM streams are written as 'pcmrecord-xxxxxx-n'. Both files are written as raw, header-less 16-bit PCM with all meta information in extended file attributes.

Most but not all modern file systems support extended attributes; a notable exception is Microsoft's VFAT32 often used as a least-common-denominator for file exchange between different operating systems. Warning! Many file copy and archiving utilities do not preserve extended attributes, at least not by default.

I/Q and PCM recordings have many uses. An I/Q recording preserves everything within the front end passband regardless of modulation type or bandwidth (provided it fits within the receiver passband). This is useful for testing and for demonstrating the 'radio' program when an antenna is not available.

I/Q recordings are especially useful as backups during critical events, such as a satellite pass or balloon flight. An I/Q recorder placed next to the SDR hardware depends only on the antenna, SDR hardware and I/Q multicast program (e.g. 'funcube'). This can guard against bugs (or the outright failure) of downstream network or processing elements, e.g., a digital data demodulator. After the demodulator is fixed, the recording can be played back to recover the data.

And of course nothing limits you to just one I/Q recording.

'iqrecord' is an excellent example of the versatility of multicasting; it can just sit quietly in the corner backing everything up without impairing any other element that might be processing the same stream.

The 'iqplay' module plays back I/Q recordings (only -- no PCM support at present) using the meta data contained in the external file attributes. It can also read a raw I/Q sample stream from standard input to simulate SDR front end hardware.

modulate

A simple (and unfinished) test modulator that takes baseband audio, amplitude modulates it on a specified carrier frequency, and emits it on standard output as an I/Q sample stream.

opus

The 'opus' module was described earlier as an optional "transcoder" that accepts uncompressed PCM (either mono or stereo) and produces a lower bit rate compressed version with the Opus codec. Options include a range of block sizes from 2.5 milliseconds to 120 ms. The larger blocks are useful only to reduce packet rate and header overhead; they provide no additional compression and are supported only by more recent versions of the Opus library. The default is 20 ms, which is long enough to give good compression but short enough to be essentially unnoticeable.

The compressed output data rate is specified by a command line parameter that defaults to 32 kb/s. This produces surprisingly good audio even on stereo music.

The Opus codec also supports a VOX-like "discontinuous" mode well suited to half-duplex push-to-talk amateur operation. When silence is detected, it automatically drops to sending "comfort noise" at only 3 packets per second even as the PCM input stream continues. If the PCM input stream stops, the Opus output stops regardless of the discontinuous setting.

Opus streams are always stereo even when the audio is mono. There is no capacity penalty and it simplifies things.

'Opus' can be run as a daemon; systemd 'service' config files are provided.

opussend

This is a standalone utility that takes PCM audio from a local sound interface, compresses it with Opus and multicasts it as an RTP network stream.

packet

This module is my first digital demodulator module for the ka9q-radio package. It accepts PCM audio from the 'radio' program, demodulates# ka9q-radio

Overview

Ka9q-radio is a collection of software defined radio (SDR) modules connected by the Internet Real Time Protocol (RTP) and IP multicasting. It is intended to facilitate experimentation and education, and to be easy to interface to data decoders and digital communication programs. It can also smoothly track satellite Doppler shift in an open-loop mode when fed velocity and acceleration information from an external program.

The ka9q-radio modules execute from the command line on either Linux or OSX. As yet there is no fancy graphical user interface. (This may come later, especially if someone volunteers to create one).

License
The License for this software is GPL3, as detailed [here](LICENSE).

Installation
Installation instructions are [here](INSTALLING.md).

Multicasting and the Real Time Protocol (RTP)

The ka9q-radio modules are designed for maximum versatility. Unlike shell pipelines or TCP/IP connections, ka9q-radio modules can be individually started and stopped at any time, and one can feed any number of others without prearrangement. They can run on the same computer or on different computers on a multicast-capable network (e.g., an Ethernet LAN).

RTP was originally designed for multicast use, though it is now used mainly in a unicast (point-to-point) mode for voice over IP (VoIP). It provides sequenced delivery, but like all real-time and multicast protocols it cannot use retransmissions to guarantee delivery. This distinguishes it from the Transmission Control Protocol (TCP) that can only be used in a point-to-point mode for data that can tolerate latency.

Ka9q-radio uses four RTP payload types: raw I/Q data with a custom receiver status header; mono or stereo 16-bit uncompressed PCM audio sampled at 48 kHz; and the new Opus codec.

It should be easy to add PCM/RTP input to any ham SDR program that uses a computer sound card to acquire receiver audio (e.g. WSJT-X, fldigi and many others). This would allow these programs to accept receive audio directly over the network from the ka9q 'radio' program, completely bypassing the computer's sound system and freeing it for other uses. This would be a much cleaner and more reliable interface as there would be no analog audio cables (real or "virtual"), "rigblaster" adapter boxes, audio ground loops, equalization problems, or tricky level adjustments! With multicasting, any number of programs can simultaneously process the same receiver output.

Parts of the ka9q-radio package are well suited to "turnkey" networked receiver applications such as receive-only APRS-to-Internet gateways and Broadcastify feeds. Service descriptions are provided for Linux udev and systemd to load the daemons at boot time.

Opus Compression and Audio Monitoring

Although PCM is best for digital decoders, it requires 800 kb/s for mono and 1.6 Mb/s for stereo. Much lower data rates suffice for human listening over WiFi or a remote Internet connection. I use the new Opus codec standardized by the Internet Engineering Task Force (IETF). An excellent open-source implementation is available, and it can be freely used without patent restrictions (unlike, e.g., ABME or AAC). Opus automatically switches algorithms as needed to do the best it can with whatever data rate it is given, from very high fidelity stereo audio at up to (an unnecessarily high) 510 kb/s down to monaural voice at only 6 kb/s. (VK5DGR's CODEC2 can handle even lower data rates, which is on my to-do list.) I typically use Opus at 32 kb/s, which is actually much higher than needed for communications-quality audio.

Popular media players like VLC support PCM and Opus so you can listen to ka9q-radio audio on many smartphones and tablets. Currently the player must receive audio multicasts, which usually means it must be on the same LAN as the computer(s) generating them. Remote unicast and tunneled multicast connectivity is on the to-do list.

The ka9q-radio package also includes 'monitor', a RTP audio player that can receive and mix several multicast audio streams to the local sound output. It has a novel 'pan' feature that uses gain and delay adjustments (<1 ms) to place each source at a user-chosen point in a stereo image. This makes it easier to distinguish multiple sources, e.g., in a round table. The 'monitor' program currently supports only Opus and PCM, though CODEC2 is again on the list.

Direct, low-latency access to multicast data on remote networks requires some form of IP multicast routing or tunneling that is not (yet) provided in this package. The remote multicast data you are most likely to want is audio, so here's a simple workaround for remote listening:

```
ssh remotesys 'pcmcat -2 pcm.vhf.mcast.local | opusenc --quiet --raw --bitrate 32 -
-' | play -q -
```

This remotely executes the 'pcmcat' command, which picks up the specified multicast group (which must be PCM audio with RTP types 10 or 11), forces the output to be stereo (which opusenc expects by default), and compresses it with the Opus codec to 32 kb/s. The compressed audio is then sent over the SSH channel to your local computer where the 'play' program decompresses and plays the audio. This requires the 'opus-tools' package on the remote system and the 'sox' package on the local system. Latency can be several seconds because of shell pipeline and TCP buffering.

There's no performance penalty to running Opus in stereo mode on mono data, so for simplicity the ka9q-radio package always uses Opus in this way.

The KA9Q 'radio' program

The heart of the ka9q-radio package is the program 'radio', an interactive general coverage receiver. It reads raw I/Q data multicast by a SDR front end, sending it tuning and gain adjustments as needed. Radio's output is a mono or stereo PCM audio multicast stream.

If audio compression is needed, a separate 'opus' module transcodes PCM to Opus. I.e., it receives PCM, compresses it with Opus and multicasts it to a different IP address. This lets listeners and players select compressed or uncompressed audio by merely joining the corresponding multicast group.

The user interface is simple but powerful. It runs in text mode, which makes it easy and fast to run remotely. If desired, the interactive interface can be disabled entirely and all parameters given instead on the command line (e.g., in shell scripts).

Architecture

The 'radio' program accepts a generic I/Q (complex) sample stream

multicast by a computer with a direct conversion ("zero IF") SDR front end. Many inexpensive SDR front ends are available, though at present only the AMSAT UK FUNcube Pro+ dongle (heretofore called the "FCD") is supported (with the 'funcube' module). The (mostly higher) sample rates produced by other receivers such as the SDRPlay and RTL-SDR are supported but only the FCD's 192 kHz sampling rate has actually been tested.

An I/Q stream at 192 kHz with 16 bit samples is a constant 6.5 Mb/s network load; this is no problem for modern Ethernet but it should be kept away from inexpensive WiFi base stations and public Internets. (More on problems with multicast and WiFi in the end notes).

Hardware Artifact Removal

Although most SDR front ends use direct conversion, the complete SDR front end/ka9q receiver combination actually forms a dual-conversion superhet: the first LO and mixer are in analog hardware and the second LO and mixer are in software. The first mixer, a quadrature (I/Q) type, produces a complex first IF centered on zero Hz and extending from -Fs/2 to +Fs/2, where Fs is the A/D sample rate. For the FCD, Fs = 192 kHz so the IF represents a "window" on the radio spectrum from 96 kHz below the first LO frequency to 96 kHz above. (The Nyquist or minimum sample rate for a complex signal is equal to the bandwidth; for a more familiar real signal, the Nyquist rate is twice the signal bandwidth.)

This is often called a "zero IF" architecture but is more accurately called a "low frequency IF". The actual first IF can be anywhere between -Fs/2 and +Fs/2 (including zero) provided it contains the entire signal bandwidth. The 10 kHz near each edge from a FCD is best avoided because of imperfect anti-alias filtering before the A/D converters, and I presume other SDR front ends have similar limitations.

All analog direct conversion SDR front ends produce a "DC spike" at 0 Hz from crosstalk from the first LO to the receiver input. On the FCD this spike is about -50 dBFS (decibels below full scale, i.e., A/D saturation). The first processing step in 'radio' completely removes it, but reciprocal mixing produces a "mound" of low frequency phase noise that cannot be removed. With the FCD this noise is only noticeable within a few kilohertz of zero and then only when the antenna is disconnected to remove external noise. But it is still easy to avoid by simply shifting the signal to a higher IF. It is common in these designs to place the IF near one quarter of the sample rate, e.g., the range from +48 kHz to +51 kHz (or -51 kHz to -48 kHz) might be selected for 3 kHz wide SSB. The 'radio' program tries to keep the IF in one of these two ranges as the user changes frequency.

48 kHz is still an extremely low first IF by usual standards, as ordinary pre-mixer filters cannot reject an image only 96 kHz away. This problem is avoided by the use of complex I/Q signals that permit the frequency image to be canceled with digital signal processing. Good image rejection requires the 'radio' program to compensate for any slight gain and phase imbalances between the I and Q channels, and this is its second processing step. Typical values for the FCD are 0.07 dB of gain imbalance and 0.4 degrees of phase error, i.e., deviation from exactly 90 degrees between the two channels.

Hardware artifact removal was recently added to the 'funcube' and 'hackrf' modules. It is still present in 'radio' but is disabled by default.

Frequency conversion

The third step is to convert the first IF signal to baseband, i.e., a carrier frequency of 0 Hz. This is performed by the second (software) LO and mixer using complex arithmetic to suppress the unwanted image. The LO and mixer each require only one complex multiply per sample. No lookup tables are used, and sine and cosine calls are needed only when changing the LO frequency.

Another LO and mixer are optionally used for open-loop correction of satellite Doppler shift. The 'radio' program reads velocity and acceleration from an external program, calculates and applies the appropriate frequency offset and rate.

When Doppler steering is active, a second LO is added to sweep the frequency of the first, requiring only two more complex multiplies per sample. Sine and cosine calls are needed only when the frequency sweep rate is changed. The Doppler steering is so smooth and accurate that I have been able to copy CW from a LEO cubesat transmitting on 70cm. Even on a near-overhead path, the signal was rock stable, staying in a narrow CW filter. (Of course, this requires accurate orbital elements and current clock time. It might be possible to use observed frequency errors to correct a set of orbital elements.)

Filtering and Demodulation

The baseband signal is filtered with fast correlation. With complex signals, the passband can be asymmetric, e.g., +100 to +3000 Hz selects upper sideband (USB) and -100 to -3000 Hz selects LSB. A post-detection frequency shift is provided for CW; this is equivalent to re-tuning the radio while simultaneously sliding the filter to keep the signal at the same point in the passband. Modes other than SSB typically use filters centered on 0 Hz but this is not required; the edges can be individually varied, e.g. to suppress adjacent channel interference.

The filtered baseband signal is then fed to one of three demodulators: 'AM', 'FM' and 'Linear.'

AM Demodulator

The AM demodulator is an ordinary non-coherent envelope detector that simply measures the amplitude of the complex baseband signal, ignoring the phase. A squelch needs to be added to make it usable for aviation communications.

FM Demodulator

The FM demodulator handles narrow-band frequency or phase modulation by taking the phase of a baseband signal with the arctangent function and then differentiating it (frequency is the derivative of phase). The arctangent instruction is surprisingly fast even on a Raspberry Pi, so a lot of literature on fast arctangent approximations seems to be unnecessary.

Two submodes are provided. 'FM' has a high pass cut on the received audio below 300 Hz to remove PL/CTCSS tones and a standard -6 dB/octave audio de-emphasis above 300 Hz. This seems to be common amateur NBFM practice.

The 'FMF' (FM Flat) mode is straight FM without post-detection filtering so the signal is treble-heavy and any PL tones are audible. Both modes use an FFT to measure PL/CTCSS tone frequencies to 0.1 Hz accuracy. SNR, frequency offset and peak deviation are also measured and displayed.

A classic FM demodulator ignores signal amplitude, but I use it in an experimental threshold extension scheme. Complex baseband samples with amplitudes falling below a certain fraction of the average signal strength are blanked. (The amplitude threshold is empirically set at 0.55 of the average). This is very effective at removing "popcorn" noise as the SNR decreases to threshold. I'd like to see how it compares to standard threshold extension techniques such as a narrow PLL.

The FM squelch works from first principles: the signal-to-noise ratio is estimated and the squelch opens when it exceeds +6 dB. This works so well that I've felt no need for a squelch adjustment. The squelch originally closed so quickly that there was no tail, but I added a short one to ensure that the tail end of a packet transmission isn't cut off.

After the squelch closes the silent PCM output (containing all 0's) continues for a few packets to flush any digital filters decoding the stream. The PCM audio stream then stops to reduce network load. The Opus transcoder, if used, also stops producing compressed packets. When the squelch reopens, the RTP marker bit is set to flag a jump in the sample count. The RTP timestamp indicates how many silent samples were suppressed. This allows programs using the input stream for timing to maintain a proper sample count.

Linear Demodulation

The 'linear' detector is used for all other modes, including SSB/CW, ISB (independent sideband) and raw I/Q. In linear mode the filter output is the audio output: both the I and Q channels for stereo modes and just the I channel for mono. Stereo is implied by I/Q and ISB. It is optional for CW/SSB, where it produces an interesting effect on headphones because of the 90 degree phase shift (Hilbert transform) between the left (I) and right (Q) channels -- the sound no longer seems to come from a point in the middle of your brain. I think this might be useful in decreasing fatigue during long contests.

The ISB mode is the same as raw I/Q except that the filter cross-adds the positive and negative spectral components to put the lower sideband on the I (left) channel and the upper sideband on Q (right).

A PLL can coherently track an AM carrier for synchronous detection, which is usually much less noisy than regular AM envelope detection. The carrier frequency offset is averaged and displayed; This is useful for calibration of the SDR TCXO against an external reference such as WWV/WWVH or the pilot carrier of a local ATSC

digital TV transmitter.

A squarer can be enabled so the PLL can track the suppressed carrier in DSB-AM and BPSK.

User Interface

The 'radio' program has a textual user interface that uses the "ncurses" library. It looks primitive by 2018 standards, but it is fully functional and very efficient over slow Internet paths. The `h` key pops up a summary of commands. The textual user interface can also be completely disabled and all receiver parameters given on the command line, e.g, for invocation from a shell script in a dedicated system.

Startup parameters are stored as files in the .radiostate directory in the user's home directory. The parameters include the input and output multicast domain names (or IP addresses), the tuning frequency and step size, the filter settings and detector mode. The correlator block size and FIR filter length can also be set here; note they cannot be changed after the program starts except by stopping and restarting it.

Invoking 'radio' with the name of a startup file loads its settings. The state of the radio can be saved to a state file with the `w` key. On normal termination the radio state is also stored as "default". It will be automatically loaded on the next invocation if no explicit name is given.

The various operating modes ("AM", "USB", etc) are defined in /usr/local/share/ka9q-radio/modes.txt. Each entry specifies a demodulator along with filter parameters, the post-detection frequency shift, AGC responses and option flags. For example, "USB" selects the "linear" demodulator, a filter passband of +100 to +3000 Hz, a zero post-detection frequency offset, an AGC attack rate of -50 dB/sec, an AGC recovery rate of +6 dB/sec, an AGC hang time of 1.1 seconds and mono output.

The textual screen display consists of the following status windows:

The "Tuning" window shows the RF, LO and IF frequencies. All can be changed by the user. If Doppler correction is active, it is also shown here but cannot be modified by the user.

The user can change frequency with the arrow keys, a Griffin Power Mate USB knob, a mouse wheel, or by typing a complete frequency into the pop up window invoked with the `f` command. The cursor indicates the digit that will change with the USB knob, mouse wheel or up and down arrow keys. The cursor can be moved to a different digit with the left and right arrow keys, and it can be moved to the next field with the TAB key. (Shift-TAB moves to the previous field.)

The "Carrier" and "Center" fields usually differ only in the linear modes when the filter edges are not symmetric around 0 Hz. The "Carrier" field shows the (suppressed) carrier frequency that comes out at zero Hz in the receiver output while the "Center" field indicates the center of the receiver passband. For example, in USB mode with a +100 to +3000 Hz frequency response, the "Center" frequency will be (100 + 3000)/2 = 1550 Hz above the carrier

frequency.

The "shift" setting shifts the entire audio output spectrum; the indicated carrier frequency appears at the selected frequency in the audio output rather than 0 Hz. This can be useful for CW operation but is not required.

The radio is most easily tuned through the "Carrier" or "Center" fields, but it can also be indirectly tuned through the First LO and IF fields. The displayed LO frequency includes the estimated TCXO frequency error and the effects of fractional-N frequency synthesis (see the end notes).

Any number of copies of 'radio' can process the same I/Q stream provided they share the same frequency range. When there is a conflict, the last radio to re-tune the SDR LO wins. The other copies will not try to fight it; without further user input they will simply wait until the LO is again within range. To avoid unintentional changes in the LO frequency, tab to the First LO field and hit the `l` key; this will underline the entry to indicate that the field is locked. The Carrier/Center frequency fields can also be locked.

The LO frequency in an I/Q recording is set at recording time and cannot be changed during playback.

The "Signal" window shows signal, noise and SNR estimates at various points in the signal path. Levels are in dBFS, decibels below full scale (A/D saturation) as there is no way to know the analog gain ahead of the A/D converters without careful calibration, which is invariably frequency dependent. These are all output-only fields.

The "Info" window shows available information on the name of the tuned channel or frequency band. For a ham band, the allowed emissions and required license class are given. These are all output-only fields, taken from /usr/local/share/ka9q-radio/bandplan.txt.

The "Filtering" window shows the pre-detection filter settings. The first four parameters are user-adjustable as part of the field sequence selected with the TAB key. Smaller Kaiser betas give sharper filters but poorer stop-band attenuation; 0 is equivalent to rectangular windowing, i.e., no windowing at all. Larger betas give wider transitions but better ultimate rejection outside the passband. The block size and FIR (which determine the FFT frequency bin size and filter delay) can only be set at startup from a state file or on the command line. [See the later discussion about sample rates and decimation.]

The "Demodulator" window is mode-specific and output-only. The AM and linear modes include a simple AGC and the relative audio gain is shown (it needs a real S-meter). The FM mode shows the frequency offset, peak deviation and PL/CTCSS tone (if present). When the selected mode and options permit the signal-to-noise ratio to be estimated, it is shown. (These estimates are done differently from the SNR estimate in the "Signal" window so they may not exactly match.)

The "Options" window selects the various options for the "Linear" demodulator. You can select an entry with the mouse or type the mode into a pop up box with the `o` command.

The output-only "SDR Hardware" window shows the nominal A/D sample rate, TCXO offset, nominal tuner frequency (uncorrected by the TCXO offset estimate), analog gain settings and gain and phase imbalance estimates.

The "Modes" window allows a predefined operating mode to be conveniently selected with the mouse.

The "I/O" window gives information on the input and output multicast network streams: IP addresses or host names; port numbers; packet, sample and error counts; and a SDR timestamp useful when playing back recorded I/Q data. It also estimates the true input A/D sample rate against the local computer time-of-day clock.

A "Debug" area at the very bottom of the screen shows version information and various test and debugging messages.

Textual user input entered through pop up menus; for example the `f` key pops up a box into which a frequency can be directly entered as `147m435` (147.435 MHz), `760k` (760 kHz) or `1g296296` (1296.296 MHz). If no letters are used to indicate the magnitude of the decimal point, a 'best guess' is made to produce a valid frequency; e.g. 14313 will give 14.313 MHz, not 14.313 kHz (which is below the FCD's range). Note that some ranges are inherently ambiguous; e.g., `500` could be either 500 kHz or 500 MHz so they should be typed with explicit decimal points.

'radio' in quiet/daemon mode

The 'radio' program can run in a totally 'quiet' mode with no user interface. This is suitable for automatic execution at boot time for fixed channel operation (e.g., receiving and reporting APRS transmissions). Two Linux systemd service descriptions are provided: 'radio34' and 'radio39'; the first creates an instance of 'radio' listening on 144.39 MHz in FM mode and the latter does the same on 144.39 MHz. Thise two instances can share the same Funcube dongle because their frequencies are only 50 kHz apart, within the 192 kHz bandwidth of the Funcube dongle.

144.39 MHz is the standard APRS frequency for North America; 144.34 MHz is the secondary APRS frequency used by WB8ELK's 15-gram 'pico tracker' for long duration balloon flights.

Other ka9q-radio Modules

Other modules in the package provide miscellaneous functions and/or start more complex planned features. None have especially complex user interfaces; most take only a few command line arguments and can run as background daemons. They are given in alphabetical order

aprs

This unfinished module accepts decoded AX.25 frames from the 'packet' module, extracts APRS position data from a selected station, computes the azimuth, elevation and range to that station from a specified point, and commands antenna rotors to point at the transmitting station. This was written specifically for tracking high altitude balloons.

aprsfeed

This module also accepts decoded AX.25 frames from the 'packet' module. It feeds those frames to the international APRS network so position reports will automatically appear on sites such as www.aprs.fi. It can be run as a background daemon.

funcube

This module takes the digital IF from a locally connected FCD and multicasts it over the local LAN. It accepts unicast commands to tune the radio and set analog gains. It will automatically reduce gain to avoid A/D saturation. Similar modules will be written for other SDR front ends such as the SDRPlay and RTL-SDR that will either talk directly to the hardware or through "shimware" such as SoapySDR.

The 'funcube' module can be automatically loaded at boot time (or at device insertion) as a background daemon by the Linux udev/systemd subsystem. A udev rule file and a systemd service file are provided. Note that the latter specifies the multicast group for the digital IF data, and this will probably require local configuration.

In daemon mode, 'funcube' writes its status to /run/funcube#/status, where '#' is the device number (starting at 0). Scripts for two devices are provided. The Raspberry Pi can support two devices, but I prefer one Pi per dongle when possible.

hackrf

This module is functionally equivalent to'funcube' except for the HackRF One. Only reception is currently supported. The HackRF supports sample rates up to 20 MHz but has only a 8-bit A/D, so by default it samples at a high speed and decimates to 192 kHz, the same as the funcube. The USB data rate and decimation processing load is too great for a Raspberry Pi, but it will run on a low-end x86 system.

Like 'funcube', 'hackrf' can be automatically loaded at boot time under Linux.

In daemon mode, 'hackrf' writes its status to /run/hackrf0/status. (Only one hackrf device is currently supported, but this can be changed.)

iqrecord and iqplay

These modules perform the functions suggested by their names. 'iqrecord' accepts multicast I/Q or PCM audio streams and writes them to disk. Gaps in the multicast stream due to lost packets or silence suppression are seeked over in a recording to maintain the correct sample count and playback timing. With a file system that supports "holes", disk blocks need not be allocated to these silent periods.

Raw I/Q streams are written into files named as 'iqrecord-xxxxxx-n' where xxxxxx is the RTP SSRC (Stream Source Identifier) and 'n' is a number incremented to avoid overwriting existing recordings. PCM streams are written as 'pcmrecord-xxxxxx-n'. Both files are written as raw, header-less 16-bit PCM with all meta information in extended file attributes.

Most but not all modern file systems support extended attributes; a notable exception is Microsoft's VFAT32 often used as a least-common-denominator for file exchange between different operating systems. Warning! Many file copy and archiving utilities do not preserve extended attributes, at least not by default.

I/Q and PCM recordings have many uses. An I/Q recording preserves everything within the front end passband regardless of modulation type or bandwidth (provided it fits within the receiver passband). This is useful for testing and for demonstrating the 'radio' program when an antenna is not available.

I/Q recordings are especially useful as backups during critical events, such as a satellite pass or balloon flight. An I/Q recorder placed next to the SDR hardware depends only on the antenna, SDR hardware and I/Q multicast program (e.g. 'funcube'). This can guard against bugs (or the outright failure) of downstream network or processing elements, e.g., a digital data demodulator. After the demodulator is fixed, the recording can be played back to recover the data.

And of course nothing limits you to just one I/Q recording.

'iqrecord' is an excellent example of the versatility of multicasting; it can just sit quietly in the corner backing everything up without impairing any other element that might be processing the same stream.

The 'iqplay' module plays back I/Q recordings (only -- no PCM support at present) using the meta data contained in the external file attributes. It can also read a raw I/Q sample stream from standard input to simulate SDR front end hardware.

modulate

A simple (and unfinished) test modulator that takes baseband audio, amplitude modulates it on a specified carrier frequency, and emits it on standard output as an I/Q sample stream.

opus

The 'opus' module was described earlier as an optional "transcoder" that accepts uncompressed PCM (either mono or stereo) and produces a lower bit rate compressed version with the Opus codec. Options include a range of block sizes from 2.5 milliseconds to 120 ms. The larger blocks are useful only to reduce packet rate and header overhead; they provide no additional compression and are supported only by more recent versions of the Opus library. The default is 20 ms, which is long enough to give good compression but short enough to be essentially unnoticeable.

The compressed output data rate is specified by a command line parameter that defaults to 32 kb/s. This produces surprisingly good audio even on stereo music.

The Opus codec also supports a VOX-like "discontinuous" mode well suited to half-duplex push-to-talk amateur operation. When silence is detected, it automatically drops to sending "comfort noise" at only 3 packets per second even as the PCM input stream continues. If the PCM input stream stops, the Opus output stops regardless of the

discontinuous setting.

Opus streams are always stereo even when the audio is mono. There is no capacity penalty and it simplifies things.

'Opus' can be run as a daemon; systemd 'service' config files are provided.

opussend

This is a standalone utility that takes PCM audio from a local sound interface, compresses it with Opus and multicasts it as an RTP network stream.

packet

This module is my first digital demodulator module for the ka9q-radio package. It accepts PCM audio from the 'radio' program, demodulates the ancient Bell 202A AFSK modem tones and decodes AX.25 frames. The decoded frames are multicast as RTP/UDP packets. The decoded frames can also optionally be displayed on the console. Otherwise the module can run as an unattended daemon.

'Packet' can be run as a daemon; a systemd 'service' file is provided.

pcmsend

This is the same as opussend, except that the output stream is uncompressed 48 kHz PCM (mono or stereo).

pcmcat

This joins a specified multicast group, which must carry uncompressed PCM audio (RTP types 10 or 11) and emits the PCM stream on standard output. This is useful for piping into an audio compressor for remote transmission.

Footnotes and Side bars

Sample Rates and Decimation

The I/Q input sample rate must be an integer multiple of the 48 kHz audio output rate. Decimation is performed as a byproduct of the fast correlator used for pre-detection filtering. Since the input FFT in a fast correlator executes at the input sample rate, CPU loading will increase. Although the CPU loading at 192 kHz is small even on a Raspberry Pi, faster and simpler decimators are in the works to support much higher sampling rates.

By default the filter decimates the FCD 192 kHz A/D input sample rate by a factor of 4 to a 48 kHz audio output. Other SDR front ends with higher sample rates will require correspondingly higher decimation ratios. Although 48 kHz may seem excessive for communications-grade audio, I don't recommend reducing it. 48 kHz is supported by nearly every audio D/A, and it still uses only 0.154% of a gigabit Ethernet link.

The Opus codec strongly prefers 48 kHz stereo even for narrow band mono voice, and there seems to be no advantage to mono or a lower sample rate (i.e., the compressed data rate is not reduced nor is audio

quality improved at a given rate.) So if the audio bit rate is a concern, just use Opus with a specified data rate; don't reduce the PCM sample rate.

Because digital demodulators should be fed uncompressed PCM, they are best run on a computer with a fast network path to the radio program so they can be given uncompressed PCM. (Audio compression works by eliminating signal components inaudible to the human ear, but which may be very important for a digital demodulator.)

Filter Tradeoffs

Simultaneously improving both filter roll off and stop-band attenuation requires a longer FIR impulse response. This requires greater latency and somewhat increased CPU loading. Currently, the filter block size and FIR length can only be specified on the command line or in the startup file, i.e., they cannot be changed without restarting the program. Note: the length of the FFT executed by the filter is equal to the sum of the block size and FIR length minus one. The default block size of 3840 corresponds to 960 samples after 4:1 decimation to 48 kHz; this is the number of samples in a 20 ms Opus frame. 3840 + 4353 - 1 = 8192, i.e., the FFT has 8k points. While the FFTW package can handle any number of points, a power of 2 is more efficient. I've found the default parameters to be suitable for most purposes.

Fractional-N Frequency Synthesizer Artifacts

Because the first LO in the FCD is in analog hardware with the usual synthesizer tuning glitches, it is re tuned only when necessary to contain the desired signal in the first IF, i.e., between -Fs/2 and +Fs/2. Otherwise tuning is with the second (software) LO that can be smoothly and instantly tuned without glitching. This is especially important when decoding digital satellite telemetry with open-loop Doppler correction.

Like most modern radios, the FCD uses fractional-N frequency synthesis with inherent restrictions on the actual set of frequencies that can be produced. Although its programming interface accepts frequencies in 1 Hz steps, the actual tuning step size varies with frequency; in the HF and VHF range it is 1000/2048 = 0.48828 Hz so it cannot produce exact 1 Hz frequency steps.

The 'radio' program works around this in two independent ways. First, the hardware synthesizer programming formulas and constants from the FCD firmware are used to determine the actual LO frequency so any discrepancy can be corrected by the second LO. Second, only frequencies that the synthesizer can exactly produce are selected, restricting the synthesizer to a relative large step size. The step size varies with frequency so the worst (largest) value of 125 Hz is always used, with the difference again absorbed by the second LO. Although these two fixes are specific to the FCD, other SDRs undoubtedly have the same problem subject to the same workarounds -- if I can get the necessary details.

Multicasting and WiFi

High speed multicast streams can cause problems with many consumer grade WiFi base stations. Although multicasting is widely used for resource discovery (especially by Apple devices), the low data rates

do not cause any problems. But a raw I/Q multicast stream from a FCD is about 6.5 Mb/s. This will cause many consumer-grade WiFi access points to roll over and die.

WiFi (especially the later versions) dynamically adapts to varying link conditions with a very wide range of transmission speeds, from only 1 Mb/s in 802.11b to over 1 Gb/s for 802.11ac. This is great for unicast traffic because the base station transmits to each client station at the fastest rate it can receive. If a client fails to acknowledge a transmission, the base station can retry at lower rates until it gets through.

The problem is that WiFi multicasts are not acknowledged, so to ensure that they reach everyone they are transmitted at a low and usually fixed data rate. Sometimes this fixed rate can be changed, but usually not. So a channel that typically operates at, say, 200 Mb/s may grind to a near halt when a lot of multicast traffic must be sent at only a few megabits/sec. If the multicast traffic arrives faster than it can be sent, the problem is obvious.

Several fixes and workarounds are available, but usually only in base stations designed for commercial environments. I have a pair of Engenius EAP600 access points at home that implement both solutions. This particular model is now discontinued but many current models from different manufacturers continue to provide them.

The first fix is IGMP Snooping. By listening in on the Internet Group Management Protocol (IGMP) messages sent by each computer to indicate the multicast groups it subscribes to, an Ethernet switch can avoid sending multicast traffic to ports where nobody wants it. A WiFi base station usually contains an internal Ethernet switch, and if it implements IGMP snooping it can avoid relaying multicast traffic to the radio channel when no one wants it.

But what if a wireless client wants the multicast traffic? The fix here is "multicast to unicast conversion". The base station keeps track (with IGMP snooping) of which wireless clients want traffic to which multicast groups, and each group member is sent its own copy of the multicast packet as a unicast at whatever speed is necessary to reach that client, which then acknowledges it like any other unicast packet. This works very well when only a few wireless clients subscribe to multicast groups and/or the links to those clients can operate at high speed, as is usually the case. The time required to send each client its own copy as a high speed unicast is still far less than the time to send a single copy as a low speed multicast.

The problem is that most consumer-grade WiFi base stations implement neither IGMP snooping or multicast-to-unicast conversion. This may eventually change thanks to the AT&T U-verse service. U-verse carries TV in high speed multicast streams (about 7 Mb/s for HDTV). The U-verse gateway implements IGMP snooping, but countless customers have run into the problems described above when they try to use their own access points.

So how to solve the problem without buying a commercial-grade access point?

One is to simply isolate the ka9q-radio modules from the rest of your network with a stand alone switch. But this is an unsatisfactory solution since it makes everything less convenient to use. Another is to buy a switch with IGMP snooping to connect the ka9q-radio modules

to the rest of your network (or at least your WiFi access point). Nearly every "smart" or "managed" switch supports IGMP snooping while unmanaged switches do not because there is no way to configure it. These managed switches are no longer expensive; the Netgear GS105Ev2, a 5-port managed switch that supports IGMP snooping, is currently $33.81 on Amazon.

Consumer grade WiFi access points can still handle moderate multicast traffic, such as an Opus-compressed audio stream. So you can still listen to receiver audio on your laptop computer if you use an IGMP-snooping switch to isolate your access point from the high rate multicast streams between the SDR front end hardware and the 'radio' program, and between the 'radio' program and the Opus transcoder.

IP Multicast itself provides another way to confine high speed streams. The Time-to-Live field in the IP header limits the number of times a packet can be routed, and this field can be configured in each ka9q-radio module that generates them. Although Ethernet switches and WiFi access points are link level devices that do not look at the IP TTL field, setting TTL=0 means that the multicast stream won't even leave the computer that generates it. It can only be received by another process on the same computer. This can keep high speed multicast I/Q streams completely off your LAN if you are willing to run the 'radio' (and 'iqrecord' and 'iqplay') programs on the same physical computer as the 'funcube' program and hardware. If the 'opus' transcoder is also run on the same system, the 'radio' program can also set TTL=0 on its PCM audio output stream. Note that this would require that any other programs that need a PCM audio input also run on the same physical machine.

Updated 8 Sept 2018, Phil Karn, KA9Q
the ancient Bell 202A AFSK modem tones and decodes AX.25 frames. The decoded frames are multicast as RTP/UDP packets. The decoded frames can also optionally be displayed on the console. Otherwise the module can run as an unattended daemon.

'Packet' can be run as a daemon; a systemd 'service' file is provided.

pcmsend

This is the same as opussend, except that the output stream is uncompressed 48 kHz PCM (mono or stereo).

pcmcat

This joins a specified multicast group, which must carry uncompressed PCM audio (RTP types 10 or 11) and emits the PCM stream on standard output. This is useful for piping into an audio compressor for remote transmission.

Footnotes and Side bars

Sample Rates and Decimation

The I/Q input sample rate must be an integer multiple of the 48 kHz audio output rate. Decimation is performed as a byproduct of the fast correlator used for pre-detection filtering. Since the input FFT in a

fast correlator executes at the input sample rate, CPU loading will increase. Although the CPU loading at 192 kHz is small even on a Raspberry Pi, faster and simpler decimators are in the works to support much higher sampling rates.

By default the filter decimates the FCD 192 kHz A/D input sample rate by a factor of 4 to a 48 kHz audio output. Other SDR front ends with higher sample rates will require correspondingly higher decimation ratios. Although 48 kHz may seem excessive for communications-grade audio, I don't recommend reducing it. 48 kHz is supported by nearly every audio D/A, and it still uses only 0.154% of a gigabit Ethernet link.

The Opus codec strongly prefers 48 kHz stereo even for narrow band mono voice, and there seems to be no advantage to mono or a lower sample rate (i.e., the compressed data rate is not reduced nor is audio quality improved at a given rate.) So if the audio bit rate is a concern, just use Opus with a specified data rate; don't reduce the PCM sample rate.

Because digital demodulators should be fed uncompressed PCM, they are best run on a computer with a fast network path to the radio program so they can be given uncompressed PCM. (Audio compression works by eliminating signal components inaudible to the human ear, but which may be very important for a digital demodulator.)

Filter Tradeoffs

Simultaneously improving both filter roll off and stop-band attenuation requires a longer FIR impulse response. This requires greater latency and somewhat increased CPU loading. Currently, the filter block size and FIR length can only be specified on the command line or in the startup file, i.e., they cannot be changed without restarting the program. Note: the length of the FFT executed by the filter is equal to the sum of the block size and FIR length minus one. The default block size of 3840 corresponds to 960 samples after 4:1 decimation to 48 kHz; this is the number of samples in a 20 ms Opus frame. 3840 + 4353 - 1 = 8192, i.e., the FFT has 8k points. While the FFTW package can handle any number of points, a power of 2 is more efficient. I've found the default parameters to be suitable for most purposes.

Fractional-N Frequency Synthesizer Artifacts

Because the first LO in the FCD is in analog hardware with the usual synthesizer tuning glitches, it is re tuned only when necessary to contain the desired signal in the first IF, i.e., between -Fs/2 and +Fs/2. Otherwise tuning is with the second (software) LO that can be smoothly and instantly tuned without glitching. This is especially important when decoding digital satellite telemetry with open-loop Doppler correction.

Like most modern radios, the FCD uses fractional-N frequency synthesis with inherent restrictions on the actual set of frequencies that can be produced. Although its programming interface accepts frequencies in 1 Hz steps, the actual tuning step size varies with frequency; in the HF and VHF range it is 1000/2048 = 0.48828 Hz so it cannot produce exact 1 Hz frequency steps.

The 'radio' program works around this in two independent ways. First,

the hardware synthesizer programming formulas and constants from the FCD firmware are used to determine the actual LO frequency so any discrepancy can be corrected by the second LO. Second, only frequencies that the synthesizer can exactly produce are selected, restricting the synthesizer to a relative large step size. The step size varies with frequency so the worst (largest) value of 125 Hz is always used, with the difference again absorbed by the second LO. Although these two fixes are specific to the FCD, other SDRs undoubtedly have the same problem subject to the same workarounds -- if I can get the necessary details.

Multicasting and WiFi

High speed multicast streams can cause problems with many consumer grade WiFi base stations. Although multicasting is widely used for resource discovery (especially by Apple devices), the low data rates do not cause any problems. But a raw I/Q multicast stream from a FCD is about 6.5 Mb/s. This will cause many consumer-grade WiFi access points to roll over and die.

WiFi (especially the later versions) dynamically adapts to varying link conditions with a very wide range of transmission speeds, from only 1 Mb/s in 802.11b to over 1 Gb/s for 802.11ac. This is great for unicast traffic because the base station transmits to each client station at the fastest rate it can receive. If a client fails to acknowledge a transmission, the base station can retry at lower rates until it gets through.

The problem is that WiFi multicasts are not acknowledged, so to ensure that they reach everyone they are transmitted at a low and usually fixed data rate. Sometimes this fixed rate can be changed, but usually not. So a channel that typically operates at, say, 200 Mb/s may grind to a near halt when a lot of multicast traffic must be sent at only a few megabits/sec. If the multicast traffic arrives faster than it can be sent, the problem is obvious.

Several fixes and workarounds are available, but usually only in base stations designed for commercial environments. I have a pair of Engenius EAP600 access points at home that implement both solutions. This particular model is now discontinued but many current models from different manufacturers continue to provide them.

The first fix is IGMP Snooping. By listening in on the Internet Group Management Protocol (IGMP) messages sent by each computer to indicate the multicast groups it subscribes to, an Ethernet switch can avoid sending multicast traffic to ports where nobody wants it. A WiFi base station usually contains an internal Ethernet switch, and if it implements IGMP snooping it can avoid relaying multicast traffic to the radio channel when no one wants it.

But what if a wireless client wants the multicast traffic? The fix here is "multicast to unicast conversion". The base station keeps track (with IGMP snooping) of which wireless clients want traffic to which multicast groups, and each group member is sent its own copy of the multicast packet as a unicast at whatever speed is necessary to reach that client, which then acknowledges it like any other unicast packet. This works very well when only a few wireless clients subscribe to multicast groups and/or the links to those clients can operate at high speed, as is usually the case. The time required to

send each client its own copy as a high speed unicast is still far less than the time to send a single copy as a low speed multicast.

The problem is that most consumer-grade WiFi base stations implement neither IGMP snooping or multicast-to-unicast conversion. This may eventually change thanks to the AT&T U-verse service. U-verse carries TV in high speed multicast streams (about 7 Mb/s for HDTV). The U-verse gateway implements IGMP snooping, but countless customers have run into the problems described above when they try to use their own access points.

So how to solve the problem without buying a commercial-grade access point?

One is to simply isolate the ka9q-radio modules from the rest of your network with a stand alone switch. But this is an unsatisfactory solution since it makes everything less convenient to use. Another is to buy a switch with IGMP snooping to connect the ka9q-radio modules to the rest of your network (or at least your WiFi access point). Nearly every "smart" or "managed" switch supports IGMP snooping while unmanaged switches do not because there is no way to configure it. These managed switches are no longer expensive; the Netgear GS105Ev2, a 5-port managed switch that supports IGMP snooping, is currently $33.81 on Amazon.

Consumer grade WiFi access points can still handle moderate multicast traffic, such as an Opus-compressed audio stream. So you can still listen to receiver audio on your laptop computer if you use an IGMP-snooping switch to isolate your access point from the high rate multicast streams between the SDR front end hardware and the 'radio' program, and between the 'radio' program and the Opus transcoder.

IP Multicast itself provides another way to confine high speed streams. The Time-to-Live field in the IP header limits the number of times a packet can be routed, and this field can be configured in each ka9q-radio module that generates them. Although Ethernet switches and WiFi access points are link level devices that do not look at the IP TTL field, setting TTL=0 means that the multicast stream won't even leave the computer that generates it. It can only be received by another process on the same computer. This can keep high speed multicast I/Q streams completely off your LAN if you are willing to run the 'radio' (and 'iqrecord' and 'iqplay') programs on the same physical computer as the 'funcube' program and hardware. If the 'opus' transcoder is also run on the same system, the 'radio' program can also set TTL=0 on its PCM audio output stream. Note that this would require that any other programs that need a PCM audio input also run on the same physical machine.

Updated 8 Sept 2018, Phil Karn, KA9Q

The Opus codec strongly prefers 48 kHz stereo even for narrow band mono voice, and there seems to be no advantage to mono or a lower sample rate (i.e., the compressed data rate is not reduced nor is audio quality improved at a given rate.) So if the audio bit rate is a concern, just use Opus with a specified data rate; don't reduce the PCM sample rate.

Because digital demodulators should be fed uncompressed PCM, they are best run on a computer with a fast network path to the radio program so they can be given uncompressed PCM. (Audio compression works by

eliminating signal components inaudible to the human ear, but which may be very important for a digital demodulator.)

Filter Tradeoffs

Simultaneously improving both filter roll off and stop-band attenuation requires a longer FIR impulse response. This requires greater latency and somewhat increased CPU loading. Currently, the filter block size and FIR length can only be specified on the command line or in the startup file, i.e., they cannot be changed without restarting the program. Note: the length of the FFT executed by the filter is equal to the sum of the block size and FIR length minus one. The default block size of 3840 corresponds to 960 samples after 4:1 decimation to 48 kHz; this is the number of samples in a 20 ms Opus frame. 3840 + 4353 - 1 = 8192, i.e., the FFT has 8k points. While the FFTW package can handle any number of points, a power of 2 is more efficient. I've found the default parameters to be suitable for most purposes.

Fractional-N Frequency Synthesizer Artifacts

Because the first LO in the FCD is in analog hardware with the usual synthesizer tuning glitches, it is re tuned only when necessary to contain the desired signal in the first IF, i.e., between -Fs/2 and +Fs/2. Otherwise tuning is with the second (software) LO that can be smoothly and instantly tuned without glitching. This is especially important when decoding digital satellite telemetry with open-loop Doppler correction.

Like most modern radios, the FCD uses fractional-N frequency synthesis with inherent restrictions on the actual set of frequencies that can be produced. Although its programming interface accepts frequencies in 1 Hz steps, the actual tuning step size varies with frequency; in the HF and VHF range it is 1000/2048 = 0.48828 Hz so it cannot produce exact 1 Hz frequency steps.

The 'radio' program works around this in two independent ways. First, the hardware synthesizer programming formulas and constants from the FCD firmware are used to determine the actual LO frequency so any discrepancy can be corrected by the second LO. Second, only frequencies that the synthesizer can exactly produce are selected, restricting the synthesizer to a relative large step size. The step size varies with frequency so the worst (largest) value of 125 Hz is always used, with the difference again absorbed by the second LO. Although these two fixes are specific to the FCD, other SDRs undoubtedly have the same problem subject to the same workarounds -- if I can get the necessary details.

Multicasting and WiFi

High speed multicast streams can cause problems with many consumer grade WiFi base stations. Although multicasting is widely used for resource discovery (especially by Apple devices), the low data rates do not cause any problems. But a raw I/Q multicast stream from a FCD is about 6.5 Mb/s. This will cause many consumer-grade WiFi access points to roll over and die.

WiFi (especially the later versions) dynamically adapts to varying link conditions with a very wide range of transmission speeds, from only 1 Mb/s in 802.11b to over 1 Gb/s for 802.11ac. This is great for

unicast traffic because the base station transmits to each client station at the fastest rate it can receive. If a client fails to acknowledge a transmission, the base station can retry at lower rates until it gets through.

The problem is that WiFi multicasts are not acknowledged, so to ensure that they reach everyone they are transmitted at a low and usually fixed data rate. Sometimes this fixed rate can be changed, but usually not. So a channel that typically operates at, say, 200 Mb/s may grind to a near halt when a lot of multicast traffic must be sent at only a few megabits/sec. If the multicast traffic arrives faster than it can be sent, the problem is obvious.

Several fixes and workarounds are available, but usually only in base stations designed for commercial environments. I have a pair of Engenius EAP600 access points at home that implement both solutions. This particular model is now discontinued but many current models from different manufacturers continue to provide them.

The first fix is IGMP Snooping. By listening in on the Internet Group Management Protocol (IGMP) messages sent by each computer to indicate the multicast groups it subscribes to, an Ethernet switch can avoid sending multicast traffic to ports where nobody wants it. A WiFi base station usually contains an internal Ethernet switch, and if it implements IGMP snooping it can avoid relaying multicast traffic to the radio channel when no one wants it.

But what if a wireless client wants the multicast traffic? The fix here is "multicast to unicast conversion". The base station keeps track (with IGMP snooping) of which wireless clients want traffic to which multicast groups, and each group member is sent its own copy of the multicast packet as a unicast at whatever speed is necessary to reach that client, which then acknowledges it like any other unicast packet. This works very well when only a few wireless clients subscribe to multicast groups and/or the links to those clients can operate at high speed, as is usually the case. The time required to send each client its own copy as a high speed unicast is still far less than the time to send a single copy as a low speed multicast.

The problem is that most consumer-grade WiFi base stations implement neither IGMP snooping or multicast-to-unicast conversion. This may eventually change thanks to the AT&T U-verse service. U-verse carries TV in high speed multicast streams (about 7 Mb/s for HDTV). The U-verse gateway implements IGMP snooping, but countless customers have run into the problems described above when they try to use their own access points.

So how to solve the problem without buying a commercial-grade access point?

One is to simply isolate the ka9q-radio modules from the rest of your network with a stand alone switch. But this is an unsatisfactory solution since it makes everything less convenient to use. Another is to buy a switch with IGMP snooping to connect the ka9q-radio modules to the rest of your network (or at least your WiFi access point). Nearly every "smart" or "managed" switch supports IGMP snooping while unmanaged switches do not because there is no way to configure it. These managed switches are no longer expensive; the Netgear GS105Ev2, a 5-port managed switch that supports IGMP snooping, is currently $33.81 on Amazon.

Consumer grade WiFi access points can still handle moderate multicast traffic, such as an Opus-compressed audio stream. So you can still listen to receiver audio on your laptop computer if you use an IGMP-snooping switch to isolate your access point from the high rate multicast streams between the SDR front end hardware and the 'radio' program, and between the 'radio' program and the Opus transcoder.

IP Multicast itself provides another way to confine high speed streams. The Time-to-Live field in the IP header limits the number of times a packet can be routed, and this field can be configured in each ka9q-radio module that generates them. Although Ethernet switches and WiFi access points are link level devices that do not look at the IP TTL field, setting TTL=0 means that the multicast stream won't even leave the computer that generates it. It can only be received by another process on the same computer. This can keep high speed multicast I/Q streams completely off your LAN if you are willing to run the 'radio' (and 'iqrecord' and 'iqplay') programs on the same physical computer as the 'funcube' program and hardware. If the 'opus' transcoder is also run on the same system, the 'radio' program can also set TTL=0 on its PCM audio output stream. Note that this would require that any other programs that need a PCM audio input also run on the same physical machine.

Updated 8 Sept 2018, Phil Karn, KA9Q
less than the time to send a single copy as a low speed multicast.

The problem is that most consumer-grade WiFi base stations implement neither IGMP snooping or multicast-to-unicast conversion. This may eventually change thanks to the AT&T U-verse service. U-verse carries TV in high speed multicast streams (about 7 Mb/s for HDTV). The U-verse gateway implements IGMP snooping, but countless customers have run into the problems described above when they try to use their own access points.

So how to solve the problem without buying a commercial-grade access point?

One is to simply isolate the ka9q-radio modules from the rest of your network with a stand alone switch. But this is an unsatisfactory solution since it makes everything less convenient to use. Another is to buy a switch with IGMP snooping to connect the ka9q-radio modules to the rest of your network (or at least your WiFi access point). Nearly every "smart" or "managed" switch supports IGMP snooping while unmanaged switches do not because there is no way to configure it. These managed switches are no longer expensive; the Netgear GS105Ev2, a 5-port managed switch that supports IGMP snooping, is currently $33.81 on Amazon.

Consumer grade WiFi access points can still handle moderate multicast traffic, such as an Opus-compressed audio stream. So you can still listen to receiver audio on your laptop computer if you use an IGMP-snooping switch to isolate your access point from the high rate multicast streams between the SDR front end hardware and the 'radio' program, and between the 'radio' program and the Opus transcoder.

IP Multicast itself provides another way to confine high speed streams. The Time-to-Live field in the IP header limits the number of times a packet can be routed, and this field can be configured in each ka9q-radio module that generates them. Although Ethernet switches and WiFi access points are link level devices that do not look at the IP

TTL field, setting TTL=0 means that the multicast stream won't even leave the computer that generates it. It can only be received by another process on the same computer. This can keep high speed multicast I/Q streams completely off your LAN if you are willing to run the 'radio' (and 'iqrecord' and 'iqplay') programs on the same physical computer as the 'funcube' program and hardware. If the 'opus' transcoder is also run on the same system, the 'radio' program can also set TTL=0 on its PCM audio output stream. Note that this would require that any other programs that need a PCM audio input also run on the same physical machine.

Updated 8 Sept 2018, Phil Karn, KA9Q