The KA9Q-Radio Package

Phil Karn, KA9Q
karn@ka9q.net

The KA9Q-Radio package demonstrates fast convolution and IP multicasting in a flexible, multichannel software defined receiver that easily scales to hundreds of channels on low cost hardware. Multicast data streams currently include the following:

1. Raw IF from SDR hardware front ends.
2. Baseband PCM.
3. Opus-compressed audio.
4. Decoded AX.25 frames.
5. Control and status data generated and consumed by the various modules.

Fast convolution uses the Fast Fourier Transform (FFT) to efficiently compute finite impulse response (FIR) filters. For all but the shortest filters it is much more efficient than direct convolution even with a hardware vector multiply-add instruction. Fast convolution takes the FFT of a signal, multiplies the spectrum by a desired frequency response, and converts it back to the time domain.[1]

Fast convolution is especially suited to large multichannel systems, as one large forward FFT can be shared by many channels, each running a small inverse FFT on different parts of the input spectrum. I have an Intel NUC i5-8260 demodulating and recording every FM channel on the 2m, 125cm and 70cm ham bands (622 total) with 60% of the CPU left over. Three Airspy R2 front ends, one for each band, acquire nearly 10 MHz each, producing a total of 60 Ms/s, 12-bit real.

The project is open source under the GPL3 license and can be found at http://www.ka9q.net/ka9q-radio.tar.xz.

Why multicast?

It's taken quite a beating in recent years, but I'm old fashioned enough to still believe in the "UNIX Philosophy": each program should do one thing and do it well, with simple interfaces that can be used in novel ways the author may not have anticipated. The UNIX 'pipeline' was a seminal IPC (inter-process communication) scheme later extended to the Internet by TCP/IP.

UNIX pipes (and TCP connections) work well for point-to-point streams, and I've used them for signal processing. But they only have two endpoints, and you might want do drive several programs just as a hardware signal source can drive several loads through a splitter. You may want to start, stop or reconfigure one module (or move it to a different computer) without restarting everything else. In a

[1] It's *almost* this simple. Some details are discussed later.

high reliability application you might run the same program on two computers, one ready to take over if the other fails.

Sender flow control in one-to-many communication is problematic because one slow receiver might bog down others. Fortunately this isn't necessary. A real time system processes data at a well defined rate, usually defined by an A/D converter clock. Buffering can handle momentary scheduling delays and jitter, but listeners simply must keep up on average. This makes sender flow control unnecessary. Only listener flow control is needed, i.e., a listener must wait for a unit of data, process it, and wait for the next unit. This simplifies things a lot.

GNU Radio already provides very flexible interconnections between signal processing modules within a single (large) program; in fact, it uses UNIX pipes internally. But I'm trying to solve a different problem: interconnecting signal processing modules that have different authors, are written in different programming languages, each with its own libraries and APIs, and run on different computers, hardware and operating systems. The Internet Engineering Task Force, the Internet protocol standards body, has (or had) a rule to standardize only "bits on wires", i.e., actual network protocols; APIs and the like were implementation details considered out of scope. Another goal was scalability. These wise choices -- standardizing just enough and no more -- led to the Internet's near-universal adoption by just about every computer, operating system and application.

IP is more flexible than GNU Radio's IPC but it is also more costly. It would be wasteful to have a UDP/IP link from an oscillator to a multiplier (mixer) and another from the mixer to a detector, for example. But it can be used quite effectively at higher levels, e.g., from an SDR front end to a software tuner/demodulator, or from a tuner/demodulator to various recording and digital decoder programs. GNU Radio itself could receive, process and/or generate multicast IP streams, as could decoding programs like FLDIGI and WSJTX without relying on kludges like "virtual audio cables". IP multicasting is especially useful for status and control messages so everyone can see what's going on.

IP multicast

IP multicasting efficiently distributes packets to a set of destinations. The sender doesn't care (and may not even know) who its listeners are or even how many there are. This is in stark contrast to the common but wasteful practice of sending a separate unicast copy of every packet to every recipient, which requires registering and tracking these recipients. The host operating systems, Ethernet switches and multicast IP routers deliver packets only to those listeners who want them. A sender sends only one copy of each packet, and they're copied as close to each listener as possible to minimize network load.

Because acknowledgments are impractical, multicast does not use TCP, the Transmission Control Protocol that provides connection-oriented sequenced byte streams to applications like http. It instead uses UDP (User Datagram Protocol). While multicast can be used for non-realtime file transfers, it is usually used for (1) resource discovery (its most common use) and (2) real-time media streams (audio, video, etc). When streaming media, UDP is often paired with RTP, the Real Time Protocol. UDP/RTP is also used on unicast streams, e.g., VoIP (Voice over IP).

Multicast is universally supported in modern operating systems. It works well on local area neworks but is stillborn in the larger Internet except for “walled gardens” like *AT&T Uverse*, which uses IP multicast over its own fiber and VDSL network for television distribution. Similar IPTV services exist in other countries.

The biggest use of multicast is resource discovery on a LAN. Apple's Bonjour suite was standardized by the IETF as the *Zeroconf* (zero configuration) protocol suite. It is widely implemented on other operating systems and many network devices. When you plug a printer into your network, this is how it automatically appears as a printer selection on your computer. Zeroconf consists of three layers: service discovery, IP address resolution and autonomous IP address assignment. You don't have to use all three.

Except for intra-home distribution within households with U-Verse, IP multicast is little used for high data rate media streams in home and small office networks (more about this later).

Organization of the KA9Q-radio package

The ka9q-radio package emphasizes modularity. One set of programs multicasts, as a RTP/UDP stream, IF signals from various SDR hardware front ends. They also transmit their status on a separate multicast group and accept commands on that same group using a subset of the control/status protocol described below for 'radio'. These programs currently include:

Program	Device	Sample Rate & Size
funcube	AMSAT UK Funcube Pro+	192 kHz, 16 bits, complex
airspy	Airspy R2	20 MHz, 12 bits, real
airspyhf	Airspy HF+	192-912 kHz, 16 bits, complex
hackrf	HackRF One	Various, depends on decimation
modulate	(Generate and modulate signal)	Various
iqplay	(Generate signal from recording)	Various

The *radio* program

This is the workhorse. It runs as a daemon under Linux *systemd*, reading its configuration file from **/etc/radio/radio-xx.conf**, where xx is the *radio* instance name. It accepts an IF stream from one of the above programs, demodulates one or more channels within those IF streams, and emits baseband PCM as multicast RTP/UDP streams. Each channel optionally sends status and accepts commands on a separate multicast group. Channels can share an output multicast group, distinguished by the RTP SSRC (Stream Source) identifier while control/status streams, if specified, must be unique. Without a control/status stream, the parameters of a channel cannot be changed without editing the configuration file and restarting the program.[2]

[2]On the VHF/UHF bands I typically assign a control/status group to only one receiver channel so I can use it manually. The rest are given specific channels and automatically assigned RTP SSRCs equal to their frequencies.

Radio uses the overlap-and-discard (aka "overlap-and-save") form of fast convolution to select and filter one or more channels within the IF stream. A forward FFT executes on the input stream at some block rate. This rate is also common to the receiver channels so it must be carefully chosen. A short block time minimizes latency and FFT cost (since smaller FFTs are cheaper per sample to compute) but also limits channel filter sharpness. Downstream processing, e.g., the automatic gain control in the linear demodulator, also operates at the block rate. Since I usually compress demodulated audio with the Opus codec, I generally pick one of its supported block sizes.[3] The forward FFTs overlap subsequent blocks by a configurable fraction, typically 20-50%. We need this because the FFT actually computes a circular convolution that "wraps around", disturbing the linear convolution. By carrying over some of the input data from each forward FFT block to the next and discarding that much from the start of each IFFT (inverse FFT) block back in the time domain, we get the linear convolution we want.[4]

One advantage of fast convolution over polyphase filtering is that each receiver channel can have an arbitrary center frequency, bandwidth and filter provided that the filter impulse response duration does not exceed the overlap in the shared forward FFT. This is not a serious problem: if every channel requires sharp filtering, the forward overlap can be increased; if only a few channels require it, they can provide their own additional filtering.

A receiver channel downconverts a signal in two steps. First, the frequency bins covering the desired signal is extracted from the forward FFT and multiplied by the desired frequency response. A brickwall filter has an infinitely long impulse response, so a Kaiser window gracefully rolls off the response to zero at the limits of the selected bins. Since the IFFT size also determines the output sample rate, this avoids aliasing. The signal is then converted back to the time domain by an inverse FFT. Because this IFFT is smaller than the (shared) forward FFT, it is faster to compute.

Since this can only shift frequency by discrete multiples of the FFT block rate[5], fine tuning is applied after conversion back to the time domain by multiplication by a relatively low frequency complex oscillator. The FM demodulator can optionally skip this step to save time.

The FM and linear demodulators

Radio provides two demodulators: FM and linear. They can be programmed manually, by the configuration file (e.g., **/etc/radio/radio-xx.conf**), or by entries in **/usr/local/share/ka9q-radio/modes.txt**.

---

[3] 2.5, 5, 10, 20, 40, 60, 80, 100 or 120 ms. I most often use 10 or 20 ms, i.e., a block rate of 100 or 50 Hz.

[4] The IF sample rate, FFT block duration and overlap together determine the number of points in the forward FFT. This is ideally a power of 2, but modern FFT packages are still quite fast as long as the block size does not contain large prime factors. FFTW3 (which I use) recommends block sizes with any number of factors of 2, 3, 5, 7 and no more than 1 factor of either 11 or 17. The Airspy R2 has a fixed 20 MHz sample rate, so a block time of 10 ms and an overlap of 20% gives a forward FFT block size of 250,000 = $2^4 5^6$.

[5] The coarse frequency shift must be an integer number of cycles during the block time of the forward FFT. For example, if the forward FFT runs every 10 ms (i.e., at 100 Hz), then the shift must be a multiple of 100 Hz. Because of the overlap, this will be some integer multiple of forward FFT bins (e.g., 2 for 50% overlap). The input and output sample rates must also be multiples of the block rate. This is in fact one of the best ways to do sample rate conversion.

FM

The FM demodulator includes a squelch based on first principles: it measures the mean and the variance of the signal amplitude, computes the SNR, and opens the squelch if it is above a threshold. To provide a little hysteresis, the opening threshold is +8dB and the closing threshold is +6 dB. This works so well that I've rarely felt the need to adjust it. When the squelch is closed, the output PCM stream stops after flushing with a configurable number of binary 0's.

To keep the signal pristine for digital modes, no de-emphasis is applied. The need for de-emphasis is flagged in the RTP stream so that consumers can apply it if desired. The *monitor* and *opus* programs automatically do this.

I've also implemented an experimental FM threshold extension scheme that works like a noise blanker on the "popcorn" noise of an FM demodulator near threshold. These pops occur when the instantaneous vector sum of the signal plus noise wraps around the origin, causing the detected phase angle (e.g., with the *carg*() or *atan2*() functions) to slip 360 degrees.[6] My scheme works by blanking or attenuating the detected signal when the instantaneous signal amplitude falls below some threshold. Empirically I've found that a threshold of 0.4 times the average amplitude is a reasonable tradeoff between letting too many pops through and taking out too much of the signal.[7] It remains to be seen whether this is any better than a well-designed PLL FM demodulator.

Broadcast stereo multiplex is decoded either with a special stereo FM mode or by feeding the composite baseband FM signal to a separate program. A similar program is available for extracting RDS data, though this is incomplete.

Linear

A single demodulator handles SSB, CW, AM, coherent AM, I/Q, etc, by simply setting the appropriate parameters. For SSB, the desired sideband is selected in the downconverter, e.g., +50 to +3000 Hz for USB or -50 to -3000 Hz for LSB[8][9] and that's it; after conversion back to the time domain, the I (in-phase or real) channel is sent to the output and the Q channel is discarded. I/Q mode is the same except that both I and Q are sent as a stereo stream. I sometimes find this helpful in reducing auditory fatigue when listening for long periods.

An envelope detector provides "true" AM. It is also possible to put the I channel on one output and the envelope-demodulated signal to the other for eventual experiments with automatic fine tuning of SSB.

The CW modes are similar to SSB except for a post-filtering frequency shift. They set a narrow filter around 0 Hz, shift the downconverter by, e.g., 500 Hz, and shift the filtered zero-IF audio by the same

---

[6]This nonlinearity is why noisy FM and noisy SSB sound different, even on gaussian (thermal) noise.

[7]This works only because the signal hasn't been limited. Dr. Andrew Viterbi taught me to never throw away any part of a signal until you've extracted everything you possibly can from it.

[8]It's remarkable how good SSB can sound on HF thanks to accurate frequency references and digital signal processing. It often sounds better than VHF/UHF FM where high pass filters cut off below 300 Hz to protect CTCSS tones.

[9]If I were king, I'd decree USB the ham standard on all bands. The only exception would be uplinks to inverting linear transponders.

amount. This lets the operator switch between, e.g., USB and CWU, with no pitch change; only the filter width and displayed carrier frequency changes.

An optional 0-Hz PLL can be enabled in any of the linear modes, though it's meaningful only when the filter response includes 0 Hz. AM is coherently demodulated by selecting the I-channel output and turning on the PLL. This locks onto the carrier and shifts it to 0 Hz. The PLL loop bandwidth is adjustable, and a lock detector enables a slow triangular sweep when the loop is unlocked. A squaring mode lets the PLL work with suppressed carrier DSB AM.[10] If one of the sidebands has interference, the filter can be asymmetrically adjusted to reject that sideband. This is handy, for example, when listening to WWV in areas with Solar Edge PV controller interference.

Automatic gain control

There are two automatic gain control blocks at opposite ends of the processing chain, serving very distinct purposes. The first is in the front end program. It monitors the average digital power at the output of the A/D converter and adjusts the available analog gain settings to keep the A/D input within range. (If the converter firmware has its own AGC it can be used instead, usually with better results.) When available, an estimate of the overall front end conversion gain (RF input terminal to digital output) is provided to *radio,* which digitally attenuates its input by the same amount[11] to maintain constant overall gain. This avoids abrupt upsets to the second AGC at the output of the linear-mode demodulator. Since there's a lag between making an analog gain change and seeing the result in the digital stream, glitches still result so the first AGC uses hysteresis to keep this from happening too often. In practice, gain changes are rare with good A/D converters and wide samples.

The second AGC is in *radio* just before the linear demodulator output; it is not used in FM. It keeps the average output level at a selected target; the hang time and recovery rate (in dB/sec) are also settable. The SNR is estimated to provide an automatic AGC threshold. E.g., when the output target is -10 dBFS[12] and the AGC threshold is -15 dB, then the average output level will be -25 dBFS on pure noise. As the signal level increases, the gain remains constant and the output increases proportionately until it reaches -10 dBFS. As the input level rises further, gain will decrease to keep the output at a constant -10 dBFS.

A noise estimator is shared by all channels for estimating SNR. It continuously averages the energy in each forward FFT frequency bin across the entire A/D bandwidth, and the bin with the lowest average energy is assumed to contain only noise common to all bins.

---

[10]There's virtually no suppressed-carrier DSB AM on the air so this hasn't been tested much. BPSK should use dedicated demodulators that do their own carrier recovery.

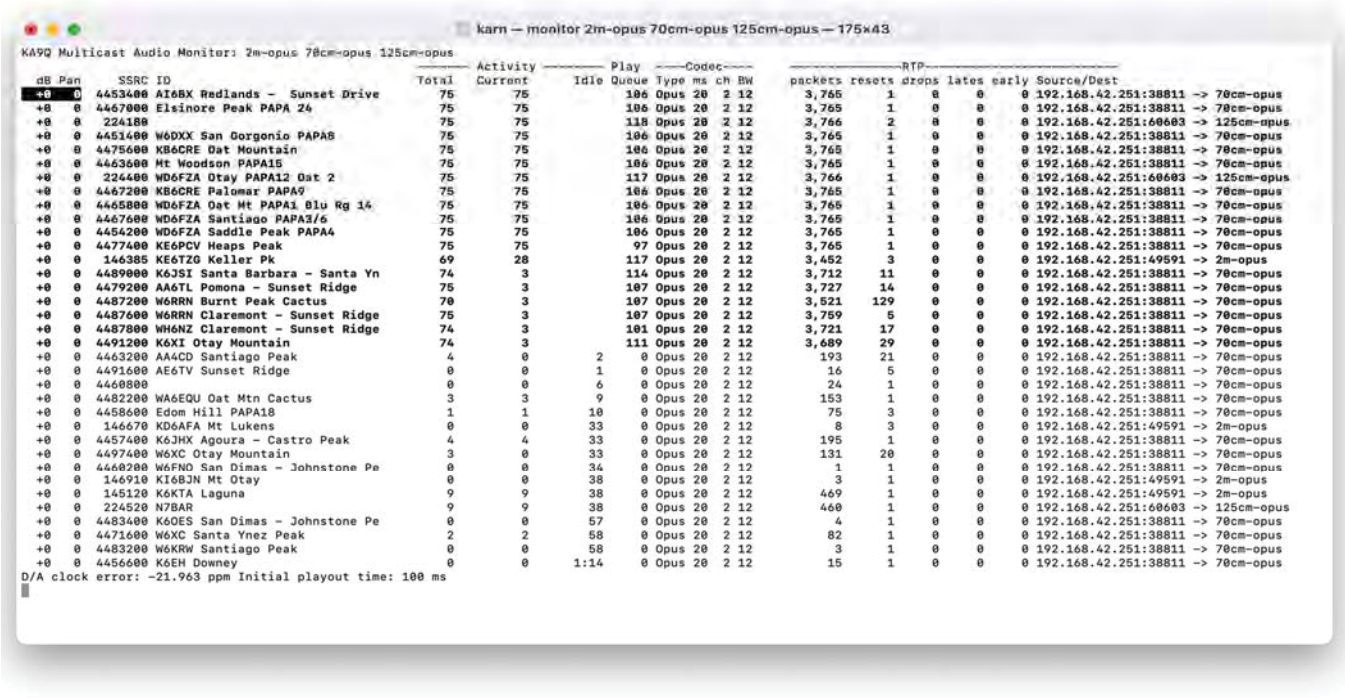
[11]All signal processing is 32-bit floating point, which has an enormous dynamic range compared to the analog signals being represented. There's no problem in running the entire receiver at unity gain right up to the output AGC stage where all the gain is applied.

[12]All signal levels in *radio* are averages over, e.g., a 10 or 20 ms block. Gaussian signals have a theoretically infinite peak-to-average ratio so it's impossible to *totally* avoid clipping and unwise to even try. But clipping is acceptably rare at an average of -10 dBFS (decibels relative to full digital scale).

Programs to process *radio's* output

The demodulated PCM output of a radio channel can be listened to with the *monitor* program, transcoded with the *opus* daemon, recorded to disk with *pcmrecord*, or demodulated with the *packet* and/or *wspr-decode* programs. Any number of programs can listen to the same stream at the same time.

The *monitor* program handles an arbitrary number of multicast groups, with an arbitrary number of distinct audio streams per group. Streams can be Opus-compressed or uncompressed PCM; a tag in the RTP header denotes the format, sample rate and channel count (mono/stereo). The user interface lets the user place each audio stream in the stereo field, adjust its level or mute it entirely.



The *pcmrecord* program records one or more audio streams. Each audio file is automatically named like this:

147075k2021-08-09T03:42:47.9Z.wav

where "147075" is the RTP SSRC (Synchronization Source Identifier, by convention the channel frequency) followed by the starting UTC date and time in ISO 8601 format.[13] Command line options specify how much silence may elapse before a file is closed. Each file represents real time, i.e., silence is removed only from the end of a file, not from within.

Because .wav files are bulky, a cron job periodically compresses older .wav files to opus. 24 kb/s is more than adequate for transparency with communications-quality voice.

The *packet* program contains a Bell 202-type demodulator and HDLC decoder. Valid packets are sent to another multicast group where they can be picked up by the *aprsfeed* and/or *aprs* programs. A single

[13]This format sorts nicely with standard UNIX commands like *ls*.

instance of *packet* handles any number of logical channels. *Aprsfeed* relays AX.25 packets to the APRS network, providing receive-only iGate functionality. The separate *aprs* command extracts position information from APRS reports and calculates azimuth, elevation and range for automatically steering an antenna; this was designed for automatically tracking high altitude balloons with APRS beacons.

The *stereo* and *rds* programs, mentioned earlier, accept composite baseband at 384 ks/s from the FM demodulator and extract stereo audio and Radio Data System data, respectively.

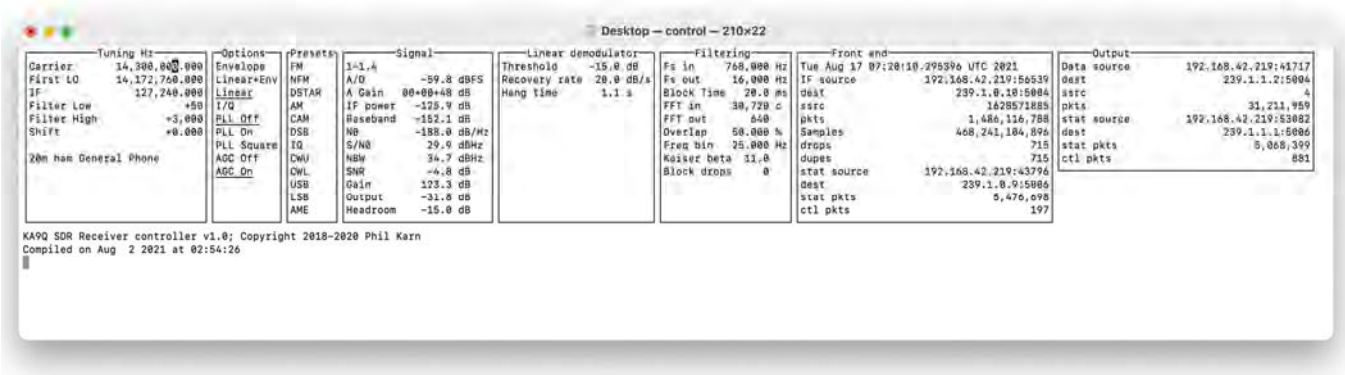
The *wspr-decode* program is similar to *pcmrecord* except that it creates a new file every two minutes and passes it to *wsprd*, the component of WSJTX that decodes received WSPR (Weak Signal Propagation Reporter). The file names expected by *wsprd* are generated.

Status/command protocol

The front end and *radio* programs use a common protocol to multicast status and accept commands. Status/command messages contain a subset of about 80 TLV (Type-Length-Value) encodings available. Each variable-length entry begins with 8-bit type and length fields so parsers can ignore unknown types. Values can be variable-length integers, IEEE single- or double-precision floats, text strings and Internet Protocol (v4 or v6) socket addresses. Some status parameters, such as radio frequency, are read/write while others (e.g., signal levels and packet counts) are read-only.

The *radio* program speaks the same protocol to its front end (as master) and to its applications (as slave). *Radio* can be omitted entirely in an application that doesn't need its features.[14]

Complete status is reported every second, and an abbreviated status is also sent every 100ms containing only changed parameters since the previous status. A flag distinguishes status from command messages. All are multicast on the same group distinct from signals so dedicated control devices don't have to receive and discard (possibly voluminous) amounts of unwanted data.



The interactive program *control* joins a specified control/status group, continually displaying status messages and sending interactive commands. Multiple copies of *control* may coexist on a status

[14]A front end looks much like an instance of *radio* with an unusually high sample rate, limited features and a fixed operating mode (e.g., I/Q or LSB). It should even be possible to concatenate instances of *radio* though I haven't tried that yet.

channel.[15] The standalone program *metadump* parses status and command messages and writes them to standard output where they can be saved and examined for debugging.

I put a lot of thought into sharing a common front end by multiple instances of *radio,* and by multiple channels within each *radio* instance. Because control and status messages are multicast, everyone follows what's going on. For example, when one *radio* channel sends a retune command to the front end (as needed at startup or in response to a explicit user command), every other channel in every instance of *radio* sees the change and automatically retunes its digital downconverter to stay on its desired radio frequency. If this isn't possible, that channel will mute its output; there will *not* be a "tuning war". Retuning is always by the minimum needed to reduce the probability of depriving other channels of their desired coverage.

Lessons learned and future work

No paper like this is complete without a candid summary of what did and didn't work, what still needs to be done, and what I'd do differently if I were starting over.

Multicasting

Multicasting is well supported by the major operating systems (I use Linux and MacOS) but its primary use is resource discovery at a few packets/sec so high rate multicast streaming is often poorly supported on low-end network hardware. Getting my home LAN[16] to properly support fast multicast streaming took some work. A "dumb" (unmanaged) switch floods multicasts to every port, which is fine until the aggregate load exceeds the slowest port speed. Then you need switches that "snoop" IGMP (Internet Group Management Protocol) or MLD (Multicast Listener Discovery, the IPv6 equivalent) so multicast traffic is only forwarded to ports with at least one listener. IGMP snooping seems a little buggy on some cheaper "smart" switches, such as my Netgear GS110TP, and it doesn't implement MLD at all. My other switches, the Linksys LGS326 and LGS528 do IGMP and MLD correctly. If I had to buy them over again, I'd start with higher grade switches.

WiFi is the real problem. Most access points send multicasts (and broadcasts) without link-level acknowledgements at some fixed, low speed that presumably reaches every client on the network and this speed stayed the same as faster modulation and coding methods were added. That's OK for low rate resource discovery, but a real time multicast stream can bring an entire WiFi network to its knees.[17] The current workaround is "multicast to unicast conversion" in the access point. It snoops IGMP or MLD and sends a separate, acknowledged unicast copy of every multicast packet to each member of a group at whatever speed that member can accept. This is usually much faster than the fixed multicast rate, so unless there are a *lot* of clients the channel loading is much lower despite the

[15]The *control* program uses *ncurses*, a 40-year-old textual windowing package that shows its age. But it works. I'd love to see others implement this protocol (or a subset) in other programs or hardware control devices, such as tuning knobs, analog meters, programmable control panels, etc.

[16]Currently a Linksys LGS326, Linksys LGS528, two Netgear GS110TPs and two Ubiquity LR access points. The APs run OpenWRT. The switches are connected with fiber to reduce RFI.

[17]Many *AT&T Uverse* users discover this when the multicast video streams cause severe problems with their existing home networks, especially WiFi.

duplication. This doesn't seem to be official in IEEE 802.11, but it is supported in the OpenWRT firmware and works well.[18][19]

But multicasting raw IF streams hasn't been as useful as I originally thought.[20] Edson, PY2SDR, wrote a nice waterfall program to display the entire IF stream, and I occasionally record or play back a raw IF stream for testing, but rarely do I have more than one instance of *radio* listening to the same IF stream. This is partly because *radio* is organized around fast convolution, the forward FFT is the “long pole in the tent” and I'm motivated to share it across as many channel threads inside *radio* as possible. When two instances of *radio* share a stream, they must duplicate the forward FFT, which I want to avoid.

I keep thinking of splitting *radio* in the middle and multicasting the raw IF in the frequency domain. Then I could move the channel downconversion and demodulation functions from threads inside *radio* to independent programs on one or more machines, which would give me a lot of flexibility. And by splitting the IF stream across multicast groups, a channel would only have to subscribe to those parts of the IF spectrum it actually needs.

The problem is network bitrate. The Airspy R2 grabs 10 MHz of spectrum (minus anti-alias filtering) and samples it at 20 MHz with 12-bit real samples. That's 240 Mb/s. My Intel NUC i5's gigabit Ethernet port now handles three such streams easily. But what about the frequency domain? *Radio* converts the Airspy's 12 bit integers to 32-bit floats, increasing the data rate to 640 Mb/s. Adding the 20-50% overlap necessary for linear convolution further expands the data rate into (and out of) the FFT to 800-1280 Mb/s. See the problem? While not every receiver need subscribe to the whole thing, the box sending it would still need a fat pipe.

A compromise, which I will probably pursue, is to put the FFT output into shared memory where at independent processes on the same machine can easily access it.[21] I can still move the downconverter/demodulator threads from *radio* into standalone programs, albeit on the same machine.

Separation of data and metadata

I separated signal data from metadata (status and commands) for several reasons: to keep high rate data away from pure control devices (they might be using WiFi), to minimize overhead on high speed streams, and so my standard RTP audio streams can be played on, e.g., *VLC*. But I haven't actually used *VLC* much; my *monitor* program has many more features, such as multiple stream support, stereo

[18]Multicast-to-unicast conversion causes an incoming multicast packet to have the client's unicast MAC address in the link-level destination field but the original multicast destination address in the IPv4 or v6 network header.

[19]What used to be an inherent property of radio – several stations receiving a single transmission – is on the way out thanks to automatic power control, cellular reuse, MIMO (multiple in, multiple out) antenna arrays and adaptive modulation and coding. Ironic, but this is the price we must pay for increased spectrum efficiency.

[20]It's still useful to connect the RF hardware to the system running *radio* with network cable rather than RF coax, and with IGMP snooping multicast costs no more than unicast. The front end program on a Raspberry Pi 4 can transfer data from an Airspy R2 to Ethernet, but so far I've been unable to get *radio* itself running in the Pi in real time; the FFT is too slow. *Radio* running on the Pi easily handles slower SDR front ends, e.g., the AirspyHF+.

[21]Linux implements threads as separate processes sharing an address space, so performance should be the same.

placement and a better (in my opinion) way of handling sample rate skew[22]. I've also defined a bunch of RTP Payload Types to represent various combinations of sample rates and de-emphasis flags, and that list is growing. I should implement the Session Descriptor Protocol (SDP), where that information really belongs.

I'm using the RTP SSRC (Synchronization Source ID) in a somewhat unconventional way. It's supposed to be random and unique to indicate the source of a particular stream. I use it to denote a particular channel multiplexed onto a multicast group, and if it's "headless" (has no control/status stream) the SSRC defaults to the fixed channel frequency. This works well. But nothing else in the RTP stream indicates the current channel frequency, and this is a problem for recordings of manually tuned channels since the SSRC is supposed to be fixed for the duration of the stream. There are lots of easy fixes if you don't care about standards, but I'd like to do things the right way.

Use of fast convolution

This project has made me a big believer in fast convolution. Aside from some tricky details, it is simple, intuitive, and much more flexible (in my opinion) than channel banks based on polyphase filtering. It may even be faster, though I haven't done the comparison myself. The heavy lifting is done by the widely used *FFTW3*[23] package, which has been tuned and optimized for many different computers. There's even a facility (*fftw-wisdom*) for tuning it to your specific machine; you can invest a lot of CPU time (once) to create a "wisdom" file that all your applications can share.

The main drawback to fast convolution is the CPU cost of the big forward FFT. It runs well on Intel x86 hardware even at high sample rates, and at lower sample rates on the Raspberry Pi, but I have yet to get it going in real time on the Raspberry Pi 4 when processing the Airspy R2 (20 Ms/s real). There's a multithreaded option but it isn't a huge win. The single forward FFT is shared by the downconversion channels so the cost is easily amortized across many of them, but it seems a little wasteful when you only have one or two. You might then be better off with conventional decimation and filtering, but it seems to me that throwing away almost all of the data from a SDR front end wastes much of its potential.

Resource discovery, configuration

As you will read below in my views on user interfaces, I believe that it should be easy to do simple, everyday things. That is not yet true for *ka9q-radio*! As with many large collections of versatile general purpose programs, a lot of configuration is needed. Once the configuration files, *udev* and *systemd*[24] service files are in place everything will come up automatically[25], but some require customization.

---

[22] *VLC* warps the playback rate, which make WWV sound really weird. I lengthen the playout buffer only when it consistently runs dry, and a manual command can reset it to a fixed value. It also resets when a stream restarts, e.g, a FM squelch opens. VLC also has a huge 1-sec playout buffer. I worked hard to minimize latency.

[23] *The Fastest Fourier Transform in the West,* version 3. https://www.fftw.org/.

[24] On Linux, *systemd* starts and manages system programs, particularly "daemons" (like *radio*) that run automatically in the background. *Udev* manages hardware devices, e.g., starting the proper front end program when a SDR device is plugged into the USB.

[25] I've been running *ka9q-radio* for years configured as a receive-only iGate (APRS message reporter) on a remote, headless Raspberry Pi 3. It requires no supervision at all except for occasional software updates.

Much work remains to be done to write these files so they need little or no editing for the common use cases, and in particular I need to learn to use the resource discovery features of Zeroconf to customize each system to its hardware. For example, my *radio* configuration files contain hardwired Airspy device serial numbers so each device can be associated with an antenna; this needs to be replaced with a "chooser" mechanism much like the one found on many computers to find and use local printers, WiFi access points, etc. I also need a mechanism to automatically assign and distribute IP multicast group addresses.

Personal views on user interfaces

I'm not a big fan of elaborate graphical user interfaces. I certainly use them but have never created one. I'm happiest designing, building and optimizing libraries, APIs, protocols and other core elements of a system. Lots of people are better at art, psychology and human design than me, and if any of you are interested in designing user interfaces, please say so!

But I do have opinions on the topic. To me, the *ideal* user interface is one that doesn't even have to exist because the program already does everything I'd ever want without being told. That's unrealistic, but at least the simple stuff should be easy. I'll go out of my way to automate something just to avoid having to create and document a user interface for it. Well-chosen defaults are critical.

I fully expect to spend a lot of effort learning the guts of a system if I want it to do something new, unusual or complex. By all means give me lots of options, test points, monitoring and debug screens, just in case I want to dig into them someday.[26] But don't make me master them all before I can do anything at all. Once I've had that little shot of dopamine from turning it on for the first time and saying "Hey, it works!" I'll soon say, "OK, now *how* does it work? What *else* can I make it do...?"

References

Mark Borgerding, "Turning Overlap-Save into a Multiband Mixing, Downsampling Filter Bank", *IEEE Signal Processing Magazine*, March 2006.
<https://www.iro.umontreal.ca/~mignotte/IFT3205/Documents/TipsAndTricks/MultibandFilterbank.pdf>

Juha Yli-Kaakinen and Markku Renfors. "Optimization of Flexible Filter Banks Based on Fast Convolution", *2014 IEEE International Conference on Acoustic, Speech and Signal Processing (ICASSIP)*.
<https://www.researchgate.net/publication/269295685_Optimization_of_Flexible_Filter_Banks_Based_on_Fast_Convolution>

Douglas L Jones, "Fast Convolution", *Connexions Project*.
<https://inst.eecs.berkeley.edu/~ee123/sp16/docs/FastConv.pdf>

[26] I've never been happy with electronic black boxes. Some of my earliest memories are of wanting to know what was behind the knobs of things like my dad's stereo system...

Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, DOI 10.17487/RFC3550, July 2003, <https://www.rfc-editor.org/info/rfc3550>.

Valin, JM., Vos, K., and T. Terriberry, "Definition of the Opus Audio Codec", RFC 6716, DOI 10.17487/RFC6716, September 2012, <https://www.rfc-editor.org/info/rfc6716>.

Spittka, J., Vos, K., and JM. Valin, "RTP Payload Format for the Opus Speech and Audio Codec", RFC 7587, DOI 10.17487/RFC7587, June 2015, <https://www.rfc-editor.org/info/rfc7587>.

Matteo Frigo and Steven G Johnson, “FFTW Home Page”, <https://www.fftw.org/>

The OpenWRT Project. <https://openwrt.org/>

Zero Configuration Networking (Zeroconf), <www.zeroconf.org>