An OS Independent and Device-Independent Mobile Web Front Panel for Radio Transceivers

Bruce Perens, Algoram
bruce@perens.com

Introduction

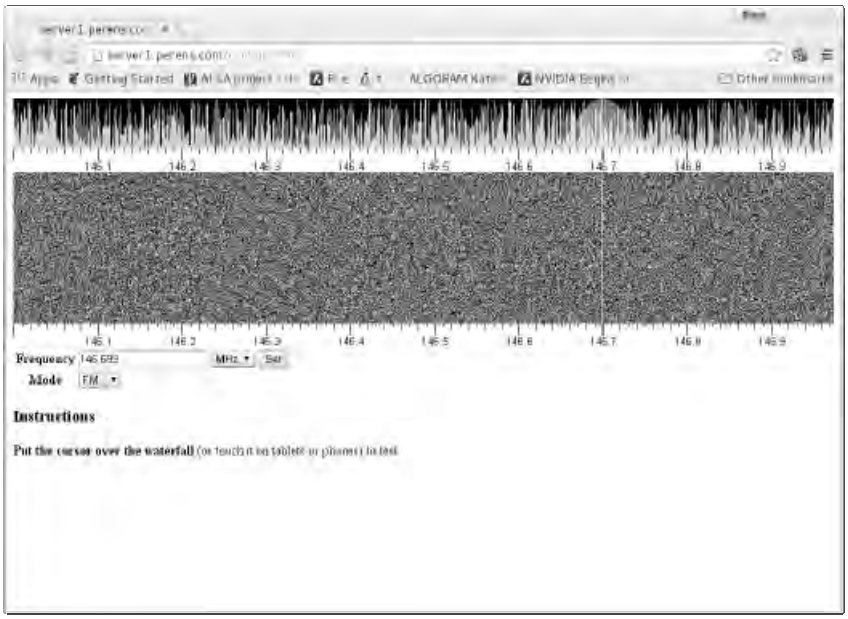

Illustration 1: A test of Algoram's waterfall bandscope using random data.

Algoram has produced a radio front panel that runs on almost all popular computer platforms, with only *iOS* as the exception at this time. This program is not *ported* from one system to another, the exact same code runs on every platform. It runs well on smartphones. The radio includes a WiFi access point and uses this means to communicate with the smartphone. Bluetooth can also be used.

The computing resources required in the radio to support this system are very modest and run in inexpensive microprocessors without virtual memory support. The smartphone interface includes a waterfall bandscope and can support all manner of graphical displays and controls. The smartphone user interface doesn't require much dexterity and can be easily used by most people. Tablets of various sizes are also supported and provide additional display area and ease-of-use.

This front panel is part of *Algoram Katena,* a 50-1000 MHz software-defined HT which can be programmed to communicate using many different modulations, modes, and protocols. We've previously refered to *Katena* as “Whitebox” or “HT of the Future.”

Katena is a front-panel-less HT which remains in the user's pocket or on a belt, and is controlled with a smartphone, with the smartphone providing the audio input and output as well as all front-panel functions. Currently this exists as a prototype larger than an HT, which will be made available in base and mobile form factors first, and then will be further miniaturized to become a handheld device.

The key to this technology has been the use of emerging HTML5 APIs to run our software in the device's web browser. There have been previous efforts that provided receiver interfaces, with bandscopes, using some form of HTML or perhaps Java. Recently, browsers have gained standardized APIs for *two-way* audio and video communications, and thus they make the microphone and camera available to the program. They were already capable of providing all manner of 2-D displays, audio output, and controls needed to operate a radio.

Revenge of the Clones

Illustration 2: A cosplayer acting as the Star Wars clone Jango Fett.

This image was created by Sam Howzit and is under Creative Commons Attribution 2.0 license. The name and image of Jango Fett are trademarked by Disney and their use here is intended to fall under the Fair Use doctrine. Downloaded from https://commons.wikimedia.org/wiki/File:Jango_Fett.jpg

Computing hardware and operating systems are fragmentary, they aren't all the same and they don't all run the same code, and this is in general a *benefit* to society. Imagine if Operating Systems were like the "clone army" in *Star Wars* episodes II and III. Just as clone troopers would all be "identical-twin brothers" who share the same DNA, Operating Systems could all look the same and run the same code. *Wouldn't that be great?*

Illustration 3: A Tasmanian Devil afflicted with contagious cancer. *Image by Menna Jones from a PLoS paper under a Creative Commons Attribution license, see https://commons.wikimedia.org wiki/File:Tasmanian_Devil_Facial_Tumour_Disease.png for details.*

No. When one clone trooper got sick, they'd probably all get sick. And here's a real-world example: because *Tasmanian Devils* became so inbred that they are "clones" from an immunological perspective, they have developed a *contagious cancer* that is driving them to extinction. Cancer isn't contagious in people because we aren't clones and we each have different immune systems.

Similarly, different software doesn't fall victim to the same viruses and security bugs at the same time, and thus a network of *hetrogenous* systems (ones different from each other) is more likely to have some portion continue to operate during an attack, while a network of *homogenous* systems (all the same) is likely to have all of its nodes fail.

Illustration 4: "The Scream" by Edward Munch, has frequently been used to illustrate frustration with computer failures, although Munch did not live in the Computer Age. *Copyright expired.*

Fifteen years ago, when the *Microsoft Windows* systems at the largest global express delivery company were attacked by the *Red Flag* virus, their entire global computer network went out of service and their hundreds of thousands of employees had to operate on phones and fax machines for a day, until of the *Windows* systems could be brought down and disinfected. Only a few systems running the *BSD* operating system maintained the company's web presence.

So, What Does This Have To Do With Ham Radio?

Hams operate *the emergency services communication network of last resort.* We are building more computers into our systems because that's the way that technology is heading, and we are creating digital networks that allow our computers to exchange data over the air, and thus make it possible for them to exchange viruses and manifest security bugs over the air. Thus, in order for our systems to be effective during emergencies, *they must not all run the same software.*

The Heartbreak of Hetrogeny

So, we've established that having *hetrogenous systems,* which don't all run the same code, are important for the security and continued operations of the world's computer networks, and are even more important to the radio amateurs who operate the network of last resort. But there is also a great *cost* to hetrogenous systems. Because they will not, in general, run the same code, we will often have to build separate programs to perform the same task on each different flavor of system.

Thus, a *native* application for an operating system, one which directly uses the CPU and its instruction set, the GUI, and the operating system services for that system, will be different from a native program on another OS or even a different hardware device. Native programs for Microsoft Windows and MacOS will in general look very different at the source-code level, and programs that look the same at the source code level will have to be recompiled for differing instruction sets, for example those on the the Intel CPUs common to desktops and the ARM CPUs common to smartphones.

So, we end up with a vast combinatorial problem. Even a company that can employ many programmers will find it difficult to economically support all of the available hardware and operating system combinations.

Software engineers have tried to solve this problem by making programs more *portable,* which means giving them the capability to be used on more than one kind of system. There have been many different approaches to solve this problem:

- We have Apple and Microsoft, who each would really prefer that everyone in the world run *their* systems so that there'd only be one kind of OS to program for and portability would not be an issue. But this brings us back to the “clone army” problem.
- We have *portability layers* like *wxWidgets* and *Qt,* which attempt to hide differences between systems at the source-code level, at the cost of an increase in program size and resource use, and the failure to include all native GUI and operating system facilities in its API.
- We have *Java*, which tries to hide the CPU, operating system, and GUI and run the same programs everywhere. This hasn't worked out as well as the Java designers would have liked, for example there is a different GUI on Android smartphones and desktops, even though both run Java, and because of differences in *Java* implementations “run everywhere” tends to have also meant “test every possible system”. Solving the performance issues of Java has taken the development of *just-in-time-compilers,* which turn Java into native code. These are large and consume their own resources.
- We have *software-as-a-service*, which runs the program on a server somewhere on the web, and provides it to the user via a web browser interface. This has the benefit of removing the need for users to administer servers and the programs that run on them, but it has tended to fail in a disaster, as the server is in general far away and must be accessed via a high-speed Internet connection. In a disaster, Internet access is likely to be interrupted.

Software-as-a-service can also have the effect of transmitting a service interruption far beyond the physical boundaries of a natural disaster. Hurricane Sandy took many data centers down, effecting software-as-a-service customers worldwide because not every service provider had, or could afford, fail-over mechanisms outside of the disaster area, which spanned several U.S. States.

- We have *computer languages,* many different kinds, which in general hide the differences between CPU instruction sets but not operating systems or GUIs. For example, the *C* language is available on very many different CPUs, and once you have a *C* compiler, you can use it to build the facilities of many other languages, for example *Python* and *Java.*

A New Hope

The web started out as a very simple way of displaying pages with links to other pages, but it didn't stay that way for long. The needs of providing additional interactivity, mostly to support software-as-a-service, inspired the implementation of *Javascript* (a different thing from *Java)* and the addition of many new APIs, and this continues to the present day.

On the one hand, this means that web programming is architecturally messier, and more difficult, than if the whole thing had been designed at the same time. Web programming now requires the use of at least *three* separate computing languages, *HTML, JavaScript, and CSS,* for the portion of a program that runs in a web browser, often a *fourth* language is used to implement the server-side software, and there can be even more languages involved, for example *SVG* which is used to define resolution-independent vector images, and *MathML* to format mathematical equations. There are also dozens of APIs to learn, as shown in the illustration. To handle all of these facilities, web browsers have gotten huge, and use substantial resources.

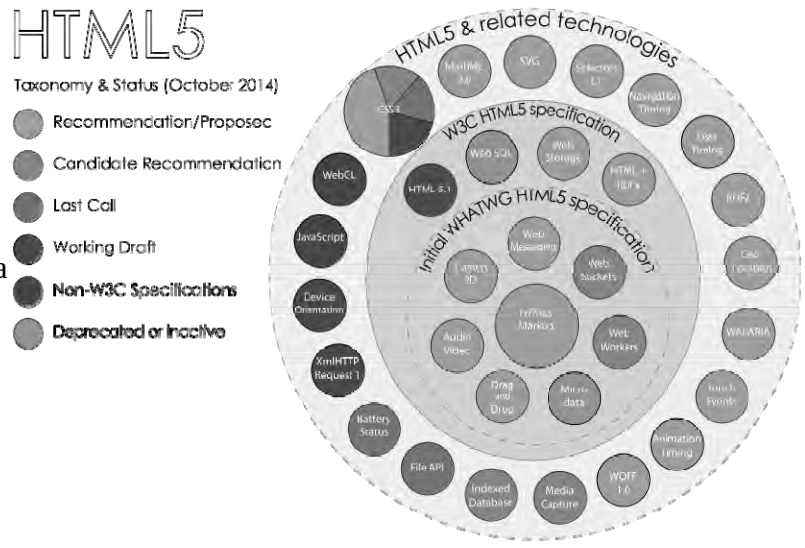

Illustration 5: HTML5 and related APIs. This image was created by Sergey Mavrody and is under the Creative Commons Attribution Share-Alike 3.0 Unported license. See https://en.wikipedia.org/wiki/HTML5#/media/File:HTML5_APIs_and_related_technologies_taxonomy_and_status.svg

On the other hand, the web browsers that closely track the HTML5 standards process, Google's *Chrome,* the Mozilla Foundation's *Firefox,* and Opera Software's eponymous browser, are now capable of all operations that we would want from a radio control panel. Most importantly, they handle two-way audio and video, 2-D graphics sufficient for radio controls and displays, and efficient network communications. These three browsers run on very many systems, and all three will run the same program. Each browser is itself built from a different code base, although some code is common to two of the three. So, there is some protection from bugs effecting all three browsers, although bugs in the user's program could be exploited on all three platforms.

The Party Pooper

Chrome, Firefox, and *Opera* aren't the only popular web browsers. There is *Safari,* which is available

in different versions on *Mac OS X* and *iOS.* Despite Apple's greater expense relative to other products and a corps of users who are perhaps fanatically dedicated to Apple and its products, Apple hasn't kept up and isn't capable of running all of the audio APIs necessary for a radio front panel without the creation of an *iOS*-specific application to support those facilities. To make the situation worse, Apple has a policy of handicapping competing browsers which it accepts for its App Store by insisting that they run Apple's own web rendering software rather than the browser developer's usual software. This means that Chrome on iOS is just as crippled by Apple's failure to keep up as Safari on iOS. This unfortunate policy doesn't exist for Mac OS X: Chrome, Firefox, and Opera are fully functional on that platform.

At this writing, it is not known at present if Apple will provide the necessary APIs on its upcoming *iOS 9.* It is expected that Apple will *eventually* provide them, but this could be years in the future.

Microsoft's *Internet Explorer* tends to have inconsistent or behind-the-times implementation of new web standards, however Chrome, Opera, and Firefox all run well on Microsoft platforms.

So, What Platforms Can We Support With HTML5 Front Panels?

At present, our HTML5 radio front panel can run on these platforms. The exact same code base runs on all of them:

- Microsoft Windows systems running *Chrome, Firefox,* or *Opera*.
- Mac OS X systems running *Chrome, Firefox,* or *Opera*.
- Android smartphones and tablets running *Chrome, Firefox,* or *Opera.*
- Linux systems running *Chrome, Firefox,* or Opera. This includes essentially all Linux distributions, for example *Ubuntu, Red Hat, Debian,* and *Centos,* but does not include *ucLinux,* which does not provide virtual memory. However, our server-side software runs on *ucLinux.*
- Chromebooks and ChromeOS.
- Kindle Fire HD 7, but only when you install *Chrome* using the *sideload* process rather than Amazon's app store.

These probably work too, or can be made to work, because they support the necessary browsers. But we've not tested them:

- Other Kindle tablets with non-e-paper displays and current OS software and the Fire phone, but only when you install *Chrome, Firefox,* or *Opera* using the *sideload* process rather than Amazon's app store.
- The BSD operating system running Chrome, Firefox, or Opera
- Firefox OS and the Firefox phone.
- Ubuntu's phone platform.

What Doesn't Work, Then?

This leaves us with *iOS* as the *only* hold-out among popular computing platforms!

And of course we could write an *app* for iOS, but that would be pandering to Apple's bad policies. We'll wait for them to catch up with web APIs.

Can We Support Even More?

Set-top boxes and TV dongles, and the various runners-up in the smartphone and tablet market: for example Microsoft's phone platform, Symbian, Blackberry, and WebOS might support, or might be persuaded to run, a browser with the required APIs. Android APIs are supported by some set-top boxes and TV dongles, and Android programs that are not directly available in the device's app store can often be *sideloaded* onto the device. We did sideload *Chrome* onto the Kindle Fire. This circumvented the artificial limitations of Amazon's app store, which declined to offer Chrome for the device in favor of Amazon's less-functional *Silk* browser.

The Operating System

Our current hardware runs ucLiunux, a compact version of the Linux operating system that runs on devices without virtual memory. We run it on an *ARM Cortex M3* CPU within the *SmartFusion II* chip, which contains our gate-array on the same die. Our CPU is a single-core 200 MHz processor, and can yet support a significant server, WiFi, Bluetooth, Ethernet, IPV4, both USB master and slave, and FLASH storage. So, we have a capable server that will fit in your pocket with the radio.

Our software is actually a form of software-as-a-service, but with the server in your pocket! Thus, there aren't the disaster-fragility problems of the usual software-as-a-service implementations. Our server remains up on battery power and communicating with local devices via WiFi and distant devices via Amateur Radio, regardless of the state of infrastructure around it.

It is expected that later devices will eventually run the full Linux system on virtual memory hardware, rather than ucLinux. The selection of Cortex M3 rather than a larger CPU is due to our use of SmartFusion II, which provides a FLASH-based gate-array which is capable of using battery power efficiently. The smilar *Igloo II* gate-array which does not provide a CPU costs as much as the SmartFusion II, so we essentially get the CPU for free!

The Web Communications APIs

At first, *WebRTC* appeared to be a desirable means of communicating between a browser and a radio. It's designed for audio and video telephony as well as data communications, and includes “NAT traversal” which solves problems with calls to systems on home networks from the outside. It's connected directly to the web audio and video APIs, and automatically scales the data compression and codecs used to make the best use of the available bandwidth.

What WebRTC lacked was a small, Open Source, embedded library that could serve it to a browser client. Our software is Open Source, and we in general prefer to use Open Source both for economic and collaboration reasons and because we *can* fix its bugs if necessary. The only Open Source software

stack available to us used a very substantial portion of the Google Chrome browser code. That was overcomplicated and would not fit in our device.

With that determined, we switched to *Websocket*, a much simpler web API for creating a data stream between a browser and another program. On the browser side, Websocket was easy to program in Javascript, requiring only a few lines of code to handle the connection.

On the server side, we made use of *libwebsockets*, a compact Open Source embedded library in the C language. This worked, and fit well in our low-resource CPU running ucLinux. Unfortunately libwebsockets was not as mature as we would have liked, and has required some debugging of its internals in order for it to work correctly in our application. This work was contributed back to the project. Since our company benefits from the work of thousands of Open Source programmers, it's only fair for us to join in that work on existing programs like libwebsockets, as well as to contribute our own new software.

The Web GUI APIs

The web GUI is built using the HTML5 *Canvas* object, and its 2-D drawing environment. This provides a Javascript API for drawing and animating all sorts of 2-D displays, buttons, and knobs, using an imaging model descended from Adobe's *PostScript.* Canvas also supports a 3-D API which we have not made use of yet.

Input comes from the keyboard, mouse, or touchscreen, and multi-touch is used to change the bandwidth of the waterfall display using a “pinch” gesture in which two fingers are used to stretch or compress the display.

Where a keyboard is available, the space bar is used for push-to-talk. On touchscreen devices we do not use push-to-talk, but a separate *transmit* and *receive* button. This works very well on smartphones. It's awkward to hold down a screen device in the way we are accustomed to holding a push-to-talk button on an HT, especially when using a smartphone. However, there is an input for a push-to-talk switch on the radio, and we can make a PTT switch available via a USB or Bluetooth peripheral. We have programmed a library to make use of USB and Bluetooth human-interface devices, and Various USB dials and pedals have been tested.

Illustration 6: A test of web push-to-talk.
Image copyrighted by the author and released under the same terms as this paper.

Web Audio

The Web Audio API is an interesting creature, providing a graph of many different audio processing nodes that can be connected to each other, including compression, gain, frequency equalization, and even a node that runs a Fast Fourier Transform. It includes a means of acquiring the system

microphone and loudspeaker and connecting them into this graph. All of this is meant to connect directly to WebRTC, which has its own nodes that work in the audio graph. Because we use Websocket, which has no such nodes, we add to this graph two nodes which process arrays of audio samples and network data in Javascript, passing the data between the web audio API and Websocket's interface to the network. Fortunately, Javascript is well-enough optimized that this runs well even on smartphones, using substantially less than the full CPU resources.

A complication of the web audio API is that it does not allow the user to select a sampling rate, but imposes its own, and this rate differs between platforms! Thus, a simple interpolation was programmed in Javascript, and is used for both audio input and output, while the audio sample rate used for network communications is set by the program. This works sufficiently well and there is still lots of CPU left, even on a smartphone.

Another complication is that Websocket does not have access to the same means of compression that would be used with WebRTC. At present we solve this by simply sending and receiving 16-bit data at 8K samples-per-second, a bandwidth that works well on Bluetooth, WiFi, and internet connections. It would be possible, although perhaps awkward, to program entire codecs in Javascript. There have been several efforts to define and implement a subset of javascript with high efficiency, which would be appropriate for codec programming.

Putting It All Together

Using all of these APIs, we have created a complete radio front panel as two programs: a client program that runs in the browser, and is coded in Javascript, HTML, and CSS, and a server program that runs in our radio device, coded in C++. The server program stores the client program on the radio, and sends it to the browser when the browser connects to the radio.

It would be awkward if it was necessary to type in a URL to connect to the radio. Indeed, the various controls of the browser aren't really necessary for our radio front panel, and our screen would be neater if we could get rid of the URL bar and the browser menus, buttons, and tabs. Fortunately, we can! There is an evolving standard for packaging web programs as Apps, which allows them to be started by touching an icon like a conventional app, and to run in full-screen mode without any of the usual browser controls visible, only the controls in our own program. Once packaged this way, web programs become indistinguishable from apps. They can be installed from an app store, or can be sent to a smartphone by our radio for direct installation.

Security

Obviously, we must not allow unauthorized individuals to control our transmitters using web interfaces. Fortunately, we have the entire set of security facilities used for other web applications: encrypted network connections over WiFi or Bluetooth, logins and passwords, etc. But this is just the start...

Authenticating Using Logbook of the World Certificates

Inspired by a paper at the TAPR conference by Heikki Hannikainen OH7LZB, and by the work of

ARRL's Logbook of the World designers, we have implemented a means for our radio to authenticate strangers on the internet as licensed radio amateurs who are allowed to control a transmitter. Individuals who have been set up by ARRL to use Logbook of the World are each sent a file containing a x.509 public-key encryption certificate. We have instructions on how to export these certificates from LoTW and load them into a web browser. Once an Amateur has done this, the browser will cryptographically authenticate itself to our radio, communicating the amateur's identity and callsign securely.

Thus, an Amateur can make the Algoram Katena radio available as a public facility on the internet, to be used by Amateurs from all countries, and can be assured that only licensed Amateurs will be allowed to connect to the radio. Since the software gives us the Amateur's call sign, it would be possible to implement a means to automatically determine what privileges an Amateur of a particular nation and license class should be granted when operating a transmitter in another nation remotely, and to disallow operation on unauthorized frequencies and modes.

In Summary

Obviously, these are features that have never existed in a walkie-talkie, and have only been partially attempted on a few experimental base stations. So, this is going to be a whole new world for Amateur Radio. The rest of our hardware and software design is equally innovative, and will be presented in