

Design and Implementation of a Portable Software Radio

Brian P. Dunn, Michael L. Dickens, and J. Nicholas Laneman

University of Notre Dame
Department of Electrical Engineering
Notre Dame, IN 46556
{bdunn2, mdickens, jnl}@nd.edu
Vox: 574-631-8034
Fax: 574-631-4393

Abstract — We describe the design and development of a portable software radio prototype device built primarily using commercial off-the-shelf components and open-source software. The device components include a general-purpose processor (GPP) on a small-form-factor motherboard, radio hardware, touchscreen and LCD, audio microphone and speaker, and an internal battery enabling hours of mobile operation. This device demonstrates that a GPP-based software radio can be implemented in a portable-form-factor using current technology. We describe the design and selection of hardware components, identification and modification of the operating system, and installation of the selected radio software. We discuss trade-offs in the selection of hardware and software, decisions that proved to be stable throughout the lifetime of the project, issues that arose, and lessons learned along the way.

1 Introduction

This article describes the design and development of a portable software radio prototype that uses as much open-source-hardware and -software as possible, and leverages commercial off-the-shelf (COTS) components. The device is shown in Figure 1, and operates using GNU Radio software for signal processing on a small-form-factor GPP-based computer and an Ettus USRP for the air interface. The prototype offers the same capabilities as GNU Radio running on a desktop computer with an Intel Core 2 Duo CPU running at 2 GHz with an Ettus USRP attached. A single device can fit inside a box approximately 29 x 10.5 x 21 cm and has roughly 2 hours of runtime off of a single battery charge. Although functionally similar software radios currently exist, they are almost entirely designed by a single manufacturer. To the best of our knowledge, this is the first portable software radio built primarily using COTS components.

The functionality of traditional hardware-based radios is limited to the capabilities present in the initial design, e.g., broadcast AM reception or analog cell-phone communications. Such devices cannot be reconfigured in any significant capacity, e.g., as an FM radio or with digital cell-phone service. An emerging architecture generally referred to as *software radio* shifts much of the signal processing into software and reprogrammable

hardware, enabling devices that can be reconfigured after deployment – including augmenting their functionality. In this article, we use the term “software radio” to refer to all types of radios that are substantially defined in software and can significantly alter their physical layer behavior through changes to their software [1, Page 2].

The concept of a software radio is credited to Dr. Joseph Mitola III in the early 1990’s [2], and refers to a class of radios that can be reprogrammed and thus reconfigured via software. The first open-source software for radios is credited to Dr. Vanu Bose in 1999 in his dissertation at MIT [3]. Dr. Bose was a member of the SpectrumWare project at MIT, which developed the original Pspetra code described in his dissertation. This code was the basis for both the Vanu Software Radio [4] – the most prominent industrial product utilizing software radio – as well as the GNU Radio open-source project [5].

Although hardware devices outperform software-based devices, the limited presence of software radios is more so an artifact of the conventional wisdom that the cost of reconfigurability is too great. There are, however, certain applications that necessitate reconfigurability, such as when device interoperability is critical, when the lifetime of a product greatly exceeds that of the devices with which it needs to communicate, or when conduct-



Figure 1. Highly reconfigurable portable software radio prototype implemented using open-source software and predominantly COTS hardware, providing dynamically configured multi-channel and full-duplex communications in most frequency bands from 50 MHz to 900MHz.

ing wireless research and development. There are also increasing numbers of applications for which software-based processing is more than adequate to meet performance requirements, and the reconfigurability provided by the use of software is highly desirable. Given the growing application-range for software radios, and the lack of industrial adoption of this technology, we set out to show that current technology is ready for, and capable of, implementing a portable-form-factor GPP-based software radio.

In the following section we give more specific objectives, motivate the decision to use GNU Radio as the software framework for the prototype, and discuss several alternative platforms that offer similar functionality. In Section 3 we detail what went into the selection and design of each hardware component along with their integration into the prototype. In Section 4 we do the same for software, discussing the lessons learned at the end of both of these sections. We conclude in Section 5 with a summary of our work on the prototype, and a discussion of some broader implications of this work and software radio as a whole.

2 Core Platform Selection

We had two primary goals. First, we wanted to create a device that could be used as a wireless research platform to quickly try out new ideas and provide a more concrete basis for what would otherwise be purely theoretical work. A key metric of success for this goal is minimizing the learning curve and development time often associated with algorithm development and experimental work in communications. Second, we set out to demonstrate that comprehensive protocol agility through software-based processing on a mobile device

is not just a technology of the future, but a viable alternative today.

Primary options for the core processor in a software radio include a field-programmable gate array (FPGA), digital signal processor (DSP), or general purpose processor (GPP). This list is roughly in decreasing order of signal processing performance, required programmer / programming specialization, and power efficiency. The list is also in increasing order of computational latency, code portability and reusability, simplicity of live-reconfiguration, and ease of processor upgradability.

Processor selection was based on the anticipated uses of the prototype, e.g., by researchers without specialized programming expertise. GPP-based software requires the least programming specialization, provides the best code-reuse, and can also be readily modified to include new or additional functionality, e.g., upgrading software from a draft to accepted wireless standard. Another argument for using a GPP is the upgrade path to newer, more capable processors via direct replacement of the processor or the motherboard on which the processor resides. A faster processor allows current code to run faster – possibly even in real-time – and for more sophisticated algorithms to be implemented, simply by recompiling for the new hardware. The advantages of using a GPP outweighed the limitations, and thus we decided to require GPP-base signal processing, but still had to decide on the hardware type – proprietary or COTS.

Proprietary hardware and software have traditionally been required when building a software radio in order to overcome some fundamental limitations including radio frequency (RF) access range, digital data-transport bandwidth, and signal processing capabilities. Commercially-available advances in the myriad radio hardware technologies – antennas and the RF front end, ADCs and DACs, data transport protocols and hardware, signal processors and small-form-factor computers, and power management systems and batteries – and the maturity of freely-available open-source radio software have significantly mitigated these limitations. Accordingly, we adopted requirements to use as much open-source software and COTS hardware as possible. The use of open-source software and a GPP for signal processing are key to both goals, by controlling device costs, providing a processor upgrade path, and allowing users to modify the original source for research purposes.

Several baseline requirements were identified to show that a GPP-based software radio could be built in a portable-form-factor offering reasonable runtime while powered from an internal battery. It was important that the system be capable of handling multiple 25 kHz voice channels with typical voice encoding, and data

transmission up to a few hundred kilobits per second with QAM, PSK, or OFDM modulation. With respect to software architecture requirements, we needed cross-platform support for Unix-like operating systems including Linux and Mac OS X, critical code written primarily in a compiled, standardized, operating-system independent programming language such as Fortran, C, or C++, and software-based control down to the physical layer. Finally, the hardware had to be economically priced, while still offering processing performance on par with currently-available desktop computers.

It was initially unclear whether we could leverage an existing software radio platform, or if it would be necessary to develop a new platform specific to our needs. To build from an established platform, we looked for mature open-source projects focusing on software-based signal processing independent of any specific operating system. There were a variety of projects that offered some of the features we needed. Those best aligned with our goals include:

- CalRadio [6]
- GNU Radio Software [5] and Ettus USRP hardware [7]
- High Performance SDR [8]
- KU Agile Radio [9]
- Rice WARP [10]
- Lyrtech Small Form Factor SDR [11]
- University of Texas HYDRA
- Virginia Tech Chameleon Radio [12]
- Virginia Tech Cognitive Engine [13]
- Virginia Tech OSSIE [14]

There were benefits and drawbacks to each of the candidate platforms. Several were too expensive while several were overly bandwidth-constrained, limiting their usefulness to voice and audio transport. Others relied exclusively on hardware for signal processing (FPGA-based) or were not sufficiently open-source. We found the combination of GNU Radio and USRP to be the best candidate, and considered the platform to be sufficiently mature to use in our devices. In the following sections we discuss the trade-offs involved with selecting this platform and describe the hardware and software development specific to the prototype. A conceptual drawing of the functionality of our chosen platform is shown in Figure 2. The discussion is structured such that readers who may have different requirements, constraints, or other considerations may readily map our decision processes and lessons learned to their environments.

3 Hardware Integration

The prototype device's hardware is comprised of a reconfigurable radio enabling communication in multiple

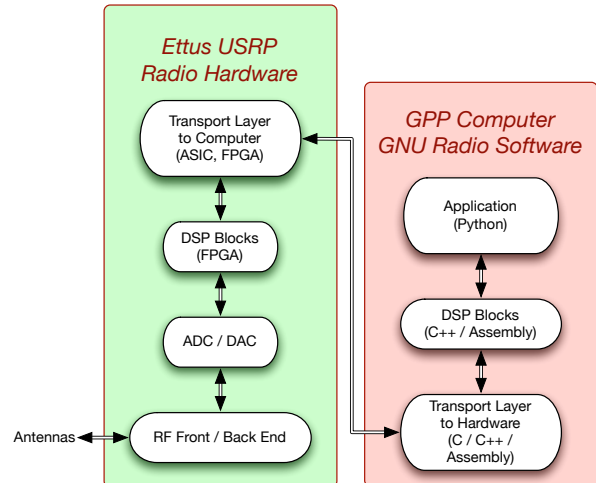


Figure 2. Block diagram depicting the functional relationship between the Ettus USRP hardware and GNU Radio software.

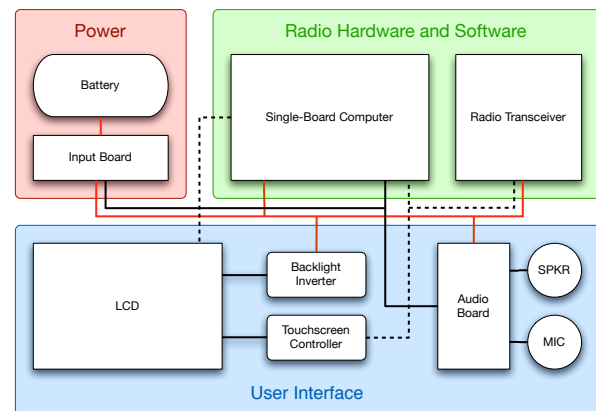


Figure 3. Block diagram of the prototype's major hardware elements, including power, LVDS, and USB connectivity.

frequency bands, a host computer to control the entire system and to perform signal processing, a touchscreen LCD and audio interface for display and user-control, and a rechargeable battery for portable operation. The block diagram in Figure 3 illustrates the system's architecture and depicts interfaces between components within the system. In the following sections, we discuss the design and integration of each hardware component and key interfaces, highlighting the challenges encountered throughout the process.

3.1 Enclosure

Three options were considered for the enclosure: sheet metal, machined aluminum, and stereolithography (SLA). SLA is the most widely-used rapid-prototyping technique for producing three-dimensional parts quickly and efficiently – the process itself takes on the order

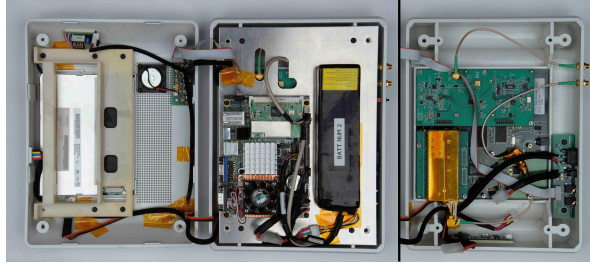


Figure 4. Internal view of the prototype showing the touchscreen LCD, backlight inverter, and audio interface on the left; the single-board computer and rechargeable battery in the middle; and, with the RF shield removed, the Ettus USRP, input board, and touchscreen controller on the right.

of hours to complete. SLA fabrication works by laser-hardening light-sensitive plastic in consecutive cross-sectional layers of the part being fabricated. Although SLA can be more expensive than some alternatives, we chose to fabricate the enclosure using SLA because a more customized enclosure could be delivered as a turn-key solution in the shortest amount of time. Two key factors in determining the device’s form-factor were the use of a commercial off-the-shelf single-board computer (SBC) and the inclusion of an LCD, both discussed below. The enclosure design is a “clam-shell” with the computer and user interface (LCD, microphone, and speaker) on the top half, and the USRP in the bottom. The open clam-shell with internal components is shown in Figure 4. The top and bottom halves are separated by a grounded sheet of aluminum that helps shield electromagnetic interference between the top and bottom of the enclosure and provides a mechanical connection for several components.

3.2 Host Computer

A number of vendors offer embedded processing boards intended for OEM integration. Taking full advantage of the existing GNU Radio software and the USRP transceiver required a SBC with a high-end chipset that was in common use. A lower-end Intel Celeron processor would have been sufficient for most applications, but an Intel Core 2 Duo offered superior performance with only a modest increase in power consumption, if any, over the Celeron. We chose the Commell LS-371 SBC because it incorporates all of the required peripherals and has one of the best performance-to-size ratios among the SBCs we evaluated. For the purposes of building a few prototypes, a commercially available SBC can provide flexible and powerful processing with little capital investment or development time. The LS-371, like most modern computers, can boot from almost any data-transport mechanism, including USB, IDE, SATA, ENET, and compact flash (CF). In order to

simplify the enclosure design and save space, we opted to use the CF memory-slot on the bottom of the board, recognizing that the throughput would likely be slower than other boot device connections.

3.3 Graphics Display

The device includes a full-resolution display and touchscreen interface that substantially enhances the platform’s functionality. However, incorporating the graphics display was the source of several unanticipated challenges, and increased the design complexity. A simpler approach would have been to use a character LCD, but that would have limited the user interface options and made on-device development overly challenging. The ideal solution was a small computer display that interfaces directly to the LS-371 and a touchscreen that emulates mouse clicks.

The main difficulties stemmed from the fact that LCDs are usually designed for a specific product, and touchscreen overlays are typically LCD-specific. Additionally, there are a variety of signaling formats used for internal video transport, further limiting what off-the-shelf display devices would work for this application. LCDs smaller than 8.4 in. usually have parallel TTL-level inputs, whereas many SBCs only provide video output over a high speed serial interface using low-voltage differential signaling (LVDS). For simplicity, we chose to use the AUO G065VN01 6.5” VGA (640x480) LCD – the smallest readily available with an onboard LVDS interface. Because the G065VN01 does not have an integrated touchscreen, we incorporated a resistive touch overlay that was designed for a similarly-sized LCD. The overlay’s output is encoded by a touchscreen controller – which are readily available using a variety of interfaces such as RS-232, PS/2, and USB. Developing the Linux software drivers for the USB controller we chose presented some additional challenges, which are further discussed in Section 4.

3.4 Power System

It was essential that the system be portable, necessitating an internal power source with enough capacity for running useful experiments in the field. However, the computationally-intensive signal processing of the SBC and USRP requires a sizable amount of power. Because weight was also of importance, heavier batteries such as lead-acid were inadequate. Lithium-ion (Li-ion) batteries, and more recent successors such as lithium-polymer (LiPo) batteries offer one of the best energy-to-weight ratios and lowest self-discharge rates available today. LiPo technology offers several additional benefits over Li-ion such as improved robustness, increased energy density, and flexible housing that enable more cus-

tomized form-factors. These benefits led to the decision to use a LiPo battery pack (with internal protection circuitry) constructed from four 3.7 V cells, which together weigh about one pound and provide a capacity of over 6 Ah at 14.8 V.

The LS-371 provides the 5 V and 12 V power supplies needed for the USRP, LCD backlight inverter, and audio amplifier. Although using the same power supply for the radio and digital boards results in increased RF noise, the overall design is much simpler and we found this solution to be acceptable for many applications. For fixed operation, an external power supply can be used via a standard 2.1 mm center pin DC jack on the back of the device.

3.5 Audio Interface

The easiest way to provide the necessary audio peripherals while interfacing with the LS-371 was to design a simple audio board specific to the prototype's needs. The audio board connects directly to the LS-371's audio header and is powered by its 5 V supply. It is mounted to the top-front of the enclosure and contains a built-in microphone, amplifier for the audio signal to an internal speaker, and logic for an externally-accessible audio port. The audio port provides 3.5 mm stereo line input and output jacks that are automatically selected when a plug is inserted or withdrawn. A low-noise adjustable gain amplifier can be switched in and out of the audio signal path to provide gain for low-level input signals, such as from an external electret microphone. All of these features are configured via an onboard DIP switch, allowing audio operation tailored for varied applications.

3.6 Lessons Learned

At the outset, it did not seem necessary to explicitly define electrical and mechanical interfaces between components, thinking that doing so would take too much time and reduce design flexibility as the project evolved. In retrospect, a more rigorous approach would have been more amenable to changes in human and capital resource allocation, allowed multiple tasks to proceed in parallel, and ultimately saved time.

Many of the challenges we encountered were caused by minor differences between our test-bench setup and the final system components. For example, something as simple as interfacing all of the components together required about fifteen different cables that collectively occupy a significant amount of space. It was difficult to recognize the importance of this issue until all of the final components were in place. The need for consistency between a test setup and the final deployment is also highlighted by problems encountered with establishing a cross-platform software development environment,

which we discuss further in the following section.

4 Software Integration

Even with the decision to use GNU Radio software for the radio, there were a number of software issues to address including selection of the operating system for the SBC, integration of drivers for hardware interfaces, and installation of the GNU Radio software and its prerequisites. This section discusses the choices and implementation of software, issues that arose and how they were resolved, and lessons learned during the integration process.

4.1 Operating System

In the spirit of keeping the project open-source, we focused on Linux for the host operating system. As the SBC we chose was quite new, we had to investigate several Linux distributions before one was found that functioned reliably. Among the free mainstream distributions that booted the SBC, Ubuntu 6.10 was the only one that functioned correctly. After choosing Ubuntu as the de-facto host operating system, we had to integrate USB-based touchscreen software and deal with boot issues created by our choice of CF storage.

4.1.1 Touchscreen Drivers

The kernel-space extension (“kext”) for USB-based touchscreens could not provide orientation parameters for our selected touchscreen; this kext is not designed for calibration. To make use of the touchscreen, we modified the USB touchscreen kext to add user-space options for swapping the X and Y coordinates and inverting the resulting X or Y axis – all independent of each other. For calibration of the incoming touchscreen data with the LCD, we chose the Evtouch X.Org event driver [15], as it was the first solution that compiled with minimal changes – even though at the time it wasn't designed specifically for Ubuntu Linux.

4.1.2 Boot Disk Issues

Compared with booting from an IDE hard drive, booting from CF was around 4 times slower at roughly 4 minutes. After reviewing the output of “dmesg” it was clear that a direct memory access timeout was stalling the boot process. A search of the particular error in the Ubuntu web forums resulted in a fix via adding the boot parameter “ide=nodma” to the GRUB “menu.lst” file for each boot command. This addition reduced the boot time to around 2.5 minutes.

4.2 Radio Software

GNU Radio provides basic building blocks for a “simple” analog repeater as well as a “complex” HDTV receiver; users can also create their own blocks. The

software package as a whole provides a framework for experimentation, testing, and evaluation of new communications protocols. Not all tasks can be done in “real-time” or “live” when using GNU Radio due to constraints inherent to the radio hardware, the host computer’s CPU, the data transport between the radio and host computer, and even the radio software itself. The demos that we developed for the prototype use GNU Radio as the underlying software framework.

There are a significant number of required packages (“pre-requisites”) that must be available before GNU Radio can function. Some of the pre-requisites are solely for compilation, while others are for runtime. Our choice of software development operating systems – Ubuntu 6.10 Linux and Mac OS X 10.4 – have different means for installing pre-requisites. As GNU Radio also requires minimum versions of some of the pre-requisites, and not all of the versions were available as pre-compiled binaries or installable packages, for both platforms some of the pre-requisites had to be installed from source. In order to ease the installation burden for each developer, we created executable scripts to handle the installation of the pre-requisites. Because we controlled what was already installed on the computers, these scripts worked reliably; on developer’s personal computers, the scripts often encountered issues due to unexpected already-installed packages.

GNU Radio software can be obtained as pre-compiled packages for some operating systems, or as an archive or repository check-out that can be compiled and installed locally. As with any rapidly-evolving open-source multi-contributor project, important bug fixes are placed in the “trunk” of the repository and not integrated into a pre-compiled package until the next release – which may be a few days, weeks, or months away. The trunk is “bleeding edge” and although it is supposed to always compile and execute properly, reality is that it doesn’t always. Hence we chose to use the trunk for the newest fixes, but to test it out with our demos before committing to a particular set of fixes. Compiling and installing GNU Radio is well-documented on their Wiki, for Ubuntu, MacOS X, and a variety of other operating systems.

4.3 Lessons Learned

During our software development efforts, we learned quite a bit about Linux, project management, software development, and participating in software projects’ lists and forums.

4.3.1 Linux Compatibility

“Linux Compatibility” in a generic sense does not necessarily mean specific Linux-distribution compatibility. Before picking a distribution, investigate any additional

required software and look for a distribution that is known to be compatible with the majority of that software.

4.3.2 Up-Front Time to Save Time Later

We wasted a lot of time waiting for the prototype to boot or reboot – which was often required during the early stages of development, especially when trying to get the touchscreen software to function. In retrospect, we should have invested more up-front time into speeding up the CF boot, and / or more thoroughly investigated other boot methods in order to reduce this waiting time.

4.3.3 Prototype Software Installs

Expect up-front to have two types of software installs: one for the *development* platform, using a fast boot interface (e.g., IDE, SATA, or USB) and another for the *testing* platform. For either storage, get a device with plenty of extra space since as software progresses it usually increases in the size of both source code and installed files.

4.3.4 Revision Control

Keeping track of the various installs of GNU Radio was difficult at best and often-times confusing. Create a local source repository with revision control (e.g., CVS, SVN, git) to hold the multiple required versions of the software being used: a *stable* branch from which most development happens; a *testing* branch with which locally-developed software can be checked for correct functionality against the latest changes of GNU Radio; and a *personal* branch for each programmer. Although the use of multiple branches in a separate repository requires some effort on all programmers’ parts, it is much easier to keep track of than each programmer maintaining a separate branch.

4.3.5 Finding and Providing Information

A major benefit of open-source software over proprietary code is the ease of access to the original source code. One can use the source code to determine how to make function calls if help files are not accurate or descriptive enough, or what a function really does via comments placed in the code, or by reviewing the code itself.

Most online software projects, and especially open-source ones, provide at least one venue in which users of all experience levels can participate – for example an email list or a web forum. These discussion venues are amazing resources for gaining knowledge about and an understanding of the project and its participants. Most are archived, and that information made available for searching.

Most software projects have multiple users, who often keep information they have learned on their personal website. Sometimes information is posted on multiple

websites, with some updated regularly while others aren't at all. While this can make gleaning information more difficult, rely on the most recent version of any information since it is the most likely to be correct.

5 Conclusions

Significant progress has been made towards making portable software radios commercially-viable, and our efforts support this endeavor. Given the ever-increasing computational power of GPPs, as well as continually increasing interest and funding for software radio and related projects, we believe that GPP-based software radio will soon provide the processing power, scalability, and reconfigurability required by today's communications problems.

The presence of an easily programmable and reconfigurable wireless platform in the research arena has the potential to accelerate innovation and stimulate more rapid deployment of new wireless protocols and platforms. Beyond the academic setting, there are several successful startup companies focusing solely on software radio. With increased attention and collaboration, we feel software radio has the potential to develop into a new technology ecosystem, similar to that emerging in the operating system market with the development of Linux by companies such as Red Hat. Coupling university research to such an ecosystem would greatly accelerate technology transfer and positively impact real-world communication systems.

6 Acknowledgments

The authors thank Phil McPhee for mechanical engineering design work, Brynnage Design

(brynnage.com) for next-day SLA fabrication of the enclosure, and our Software Defined Radio Group for their continuing efforts, including Neil Dodson, Andrew Harms, Ben Keller, Marcin Morys, and Yaakov Sloman.

This work has been supported in part by the US National Institute of Justice (NIJ) through grant 2006-IJ-CX-K034 and the US National Science Foundation (NSF) under grant CNS06-26595.

References

- [1] Jeffrey H. Reed, *Software Radio: A Modern Approach to Radio Engineering*, ser. Prentice Hall Communications Engineering and Emerging Technologies Series. Prentice Hall PTR, May 2002.
- [2] Mitola, J., III, "Software Radios - Survey, Critical Evaluation and Future Directions," *IEEE National Telesystems Conference, NTC-92*, pp. 13/15-13/23, 19-20 May 1992.
- [3] Bose, Vanu G., "Design and Implementation of Software Radios Using a General Purpose Processor," Ph.D. dissertation, MIT, 1999.
- [4] "Vanu Software Radio." [Online]. Available: www.vanu.com
- [5] "GNU Radio Software." [Online]. Available: www.gnuradio.org/trac
- [6] "CalRadio." [Online]. Available: calradio.calit2.net
- [7] "Ettus USRP Hardware." [Online]. Available: www.ettus.com
- [8] "High Performance SDR." [Online]. Available: hpsdr.org
- [9] "KU Agile Radio." [Online]. Available: agileradio.ittc.ku.edu
- [10] "Rice WARP." [Online]. Available: warp.rice.edu
- [11] "Lyrtech Small Form Factor SDR." [Online]. Available: www.lyrtech.com
- [12] "Virginia Tech Chameleon Radio." [Online]. Available: www.ece.vt.edu/swe/chamrad
- [13] "Virginia Tech Cognitive Engine." [Online]. Available: www.cognitiveradio.wireless.vt.edu
- [14] "Virginia Tech OSSIE." [Online]. Available: ossie.wireless.vt.edu/trac
- [15] "Evtouch X.Org Event Driver." [Online]. Available: www.conan.de/touchscreen/evtouch.html