Design and Implementation of a Portable Software Radio

Michael L. Dickens, Brian P. Dunn, and J. Nicholas Laneman
University of Notre Dame, Department of Electrical Engineering
Notre Dame, IN
{mdickens, bdunn2, jnl}@nd.edu
http://sdr.nd.edu

Abstract

We summarize the design and development of a portable software radio prototype built primarily using commercial off-the-shelf components and open-source software. Our research group leverages these prototypes for several funded projects focusing on issues including interoperable public safety communications, cognitive wireless networks, and educational initiatives. 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. We describe the selection of hardware and software components, identification and modification of the operating system, and development of an application-programming framework that augments 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. Significant advances over the past decade have made GPP-based software radio a viable solution in many areas, and this work demonstrates that todays processors are capable of enabling a new generation of software radio in portable form factor devices.

I. 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 [1] for signal processing on a small-form-factor general-purpose processor (GPP)-based computer and an Ettus USRP (Universal Software Radio Peripheral) [2] for the air interface. The prototype offers the same capabilities as GNU Radio running on an Intel Core 2 Duo CPU-based computer running at 2 GHz with an Ettus USRP attached. The device can fit inside a box of dimensions 29 x 10.5 x 21 cm, weighs just under seven pounds, and has roughly two hours of runtime from a single battery charge. The bill of materials for construction of a single device using retail components comes to about $3,700. The prototype described here exemplifies the benefits and cost savings offered by leveraging open-source and COTS components, and to the best of our knowledge represents the first portable software radio of its kind.

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 [3].

The concept of a software radio is credited to Dr. Joseph Mitola III in the early 1990's [4], and refers to a class of radios that can be reprogrammed and thus reconfigured via software. The first working software radio, called SPEAKeasy [5], was developed for the US Army between 1992 and 1995, but the software was not portable beyond the hardware for which it was developed and the overall system filled up the entire rear of a transport vehicle. The first open-source software for radios is credited to Dr. Vanu Bose in his 1999 dissertation at the Massachusetts Institute of Technology (MIT) [6]. Dr. Bose was a member of the SpectrumWare project at MIT, which developed the original Pspectra code described in his dissertation. This code led to both the Vanu

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

Software Radio [7] – an industrial product for cellular base stations, and the first FCC-approved software radio [8] – as well as the GNU Radio open-source project.

Software radio has found uses in amateur radio, academia, industry, government, and military applications. In communications research laboratories, software radio is being used for testing and evaluation of multihop and mobile multimedia broadcasting transmissions, wireless network experimentation and cross-layer prototyping, and so-called "cognitive" radio research and development. Software radio has also been used for data-acquisition in fields including magnetic resonance force microscopy and aeronautical testing. In all cases, the key to this technology's success is reconfigurability via software modification, providing greater flexibility and long-term cost savings over traditional hardware-only approaches. Many of these applications were developed using GNU Radio, and hence could be made to run on our prototype.

For high-volume applications, hardware radios are often preferred over software-based imple-

mentations. However, as processing performance and power efficiency increase there will be more applications for which software radios are more capable and can be built more cost effectively than their hardware counterparts. Even today, there are many applications that benefit from the superior reconfigurability offered by software-based implementations, 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, and for wireless research and development. There is also an increasing number of applications for which software-based processing is a feasible alternative and the added reconfigurability is highly desirable. Given the growing application-range for software radios, and their relatively limited presence in industry, we set out to demonstrate that today's GPPs are capable of enabling portable communication devices that offer superior flexibility and advanced functionality.

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 III we detail what went into the selection and design of each hardware component along with their integration into the prototype. In Section IV we do the same for software, discussing the lessons learned at the end of both of these sections. We conclude in Section V 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.

II. Core Platform Selection

At the early stages of the project, 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 was 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 only a technology of the future, but a viable alternative today.

A generic software radio consists of the radio hardware, optional user interfaces, and – most importantly – one or more signal processors. The primary options for each signal processor include a field-programmable gate array (FPGA), a digital signal processor (DSP), and a GPP.

We based processor selection on the anticipated uses of the prototype, e.g., by researchers

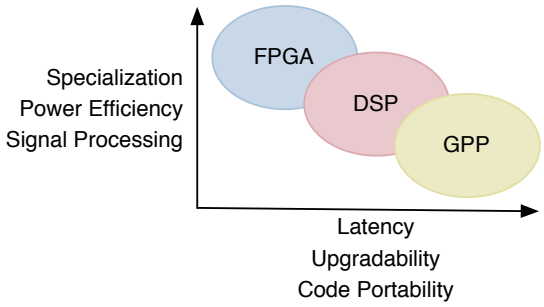

Fig. 2. Graphical depiction of the tradeoff between various properties of signal processors.

without specialized programming expertise. As shown in Figure 2, 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 execute 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 whether the hardware in which the GPP resides would be made in house or purchased COTS.

Proprietary hardware and software has 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 RF front ends, analog-to-digital and digital-to-analog converters, 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 a GPP and open-source software for signal processing are key to achieving both goals by controlling hardware costs, creating a highly scalable processor upgrade path, and fostering a collaborative environment for software development in the wireless

community.

We identified several baseline requirements 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. For our grant from the US National Institute of Justice, 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 desired 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 C++, and software-based control down to the physical layer. Finally, we wanted the hardware 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 included:

- GNU Radio software and Ettus USRP hardware
- CalRadio (`http://calradio.calit2.net`)
- High Performance SDR (`http://hpsdr.org`)
- KU Agile Radio (`http://agileradio.ittc.ku.edu`)
- Rice WARP (`http://warp.rice.edu`)
- Lyrtech Small-Form-Factor SDR (`http://www.lyrtech.com`)
- University of Texas HYDRA (`http://users.ece.utexas.edu/~rheath/research/prototyping/mimoadhoc`)
- Virginia Tech Chameleonic Radio (`http://www.ece.vt.edu/swe/chamrad`)
- Virginia Tech Cognitive Engine (`http://www.cognitiveradio.wireless.vt.edu`)
- Virginia Tech OSSIE (`http://ossie.wireless.vt.edu/trac`)

We determined benefits and drawbacks to each of the candidate platforms. Several were too expensive or overly bandwidth-constrained, limiting their usefulness to voice and audio transport. Others relied exclusively on FPGA-based signal processing or did not provide sufficiently open-

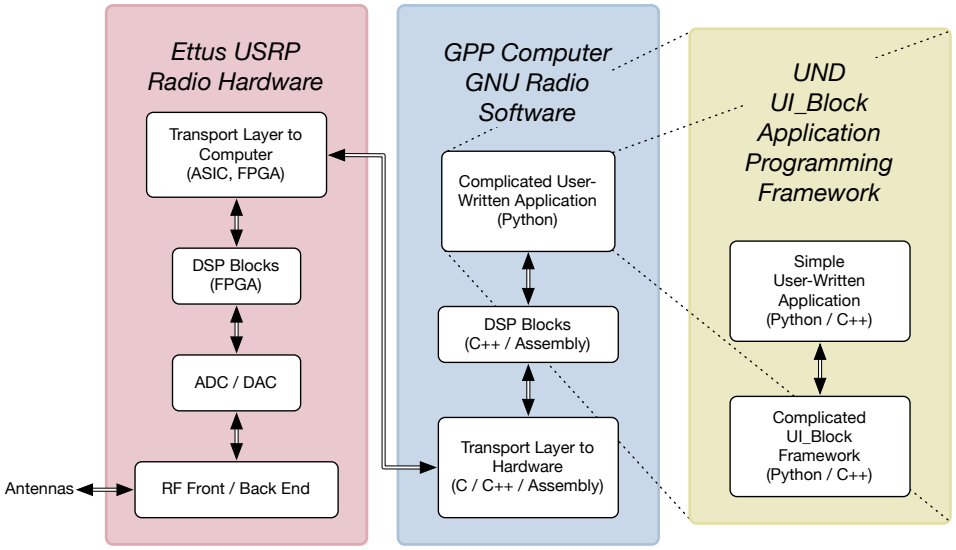

Fig. 3. Block diagram depicting the functional relationship between the Ettus USRP hardware and GNU Radio software, augmented with our user interface framework.

source software. We found the combination of the GNU Radio software and Ettus USRP hardware to be the best candidate, and considered the platform to be sufficiently mature to use in our devices. In the following sections we discuss some trade-offs involved in selecting this platform and describe the hardware and software development specific to the prototype. Figure 3 shows a conceptual drawing of the complete software radio platform and each functional component. 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.

III. Hardware Integration

Even with our decision to use the Ettus USRP for the radio hardware, there were still a number of issues we had to address including creation of the enclosure and audio interface, and selection of the host computer, graphics display, and power system. The prototype device's hardware is comprised of a reconfigurable radio enabling communication in multiple 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.

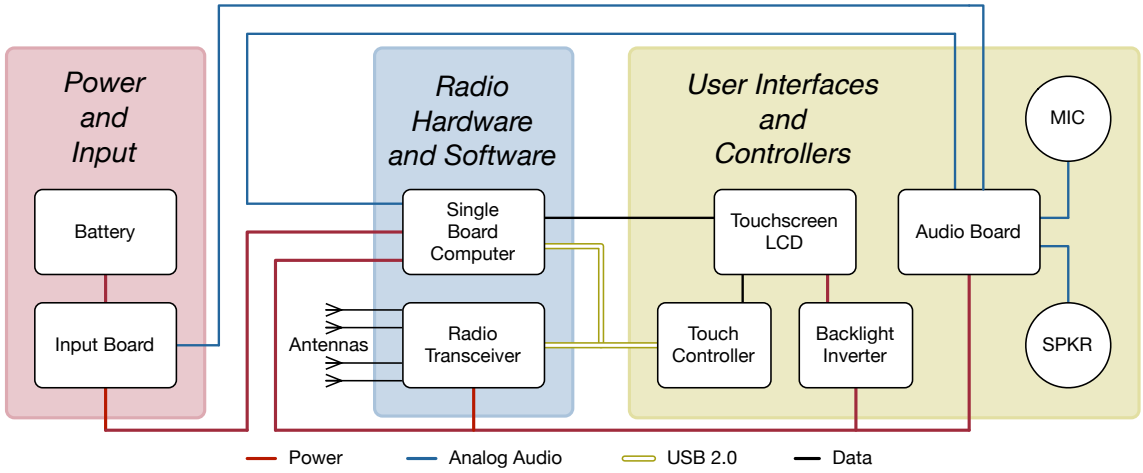

Fig. 4. High-level block diagram of the prototype's major hardware elements, including power, audio, USB, and other data connectivity.

The block diagram in Figure 4 illustrates the system's basic architecture and depicts high-level 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.

A. *Enclosure*

We considered three options for the enclosure: sheet metal, machined aluminum, and stereo-lithography (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 of hours to complete. SLA fabrication works by laser-hardening light-sensitive plastic in consecutive cross-sectional layers of the part being fabricated, followed by minor cosmetic finishing. 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 turnkey solution in the shortest amount of time.

Two key factors in determining the device's form factor were the use of a COTS 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 Ettus USRP in the bottom half. Figure 5 shows the open clam-shell with internal

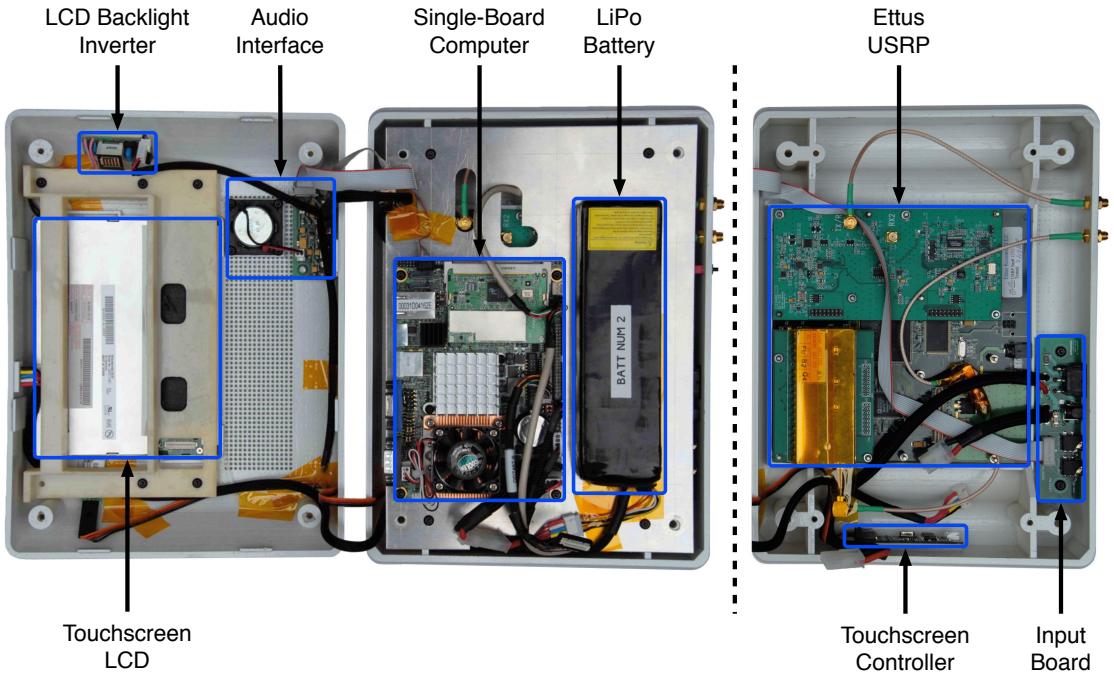

Fig. 5. Internal view of the prototype showing the touchscreen LCD, backlight inverter, and audio interface on the left; the single-board computer and rechargeable battery in the middle; and, with the RF shield removed, the Ettus USRP, input board, and touchscreen controller on the right.

components. A grounded sheet of aluminum separates the top and bottom halves of the enclosure. This sheet helps shield electromagnetic interference (EMI) between the two halves and provides a mechanical connection for several components. The EMI shield is a critical component for reliable operation of the radio; without adequate isolation the USRP picks up too much EMI from the SBC, dramatically degrading performance.

B. Host Computer

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. A number of vendors offer embedded processing boards intended for OEM integration. Taking full advantage of the existing GNU Radio software and Ettus 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 has one of the best performance-to-size ratios among the SBCs we evaluated, and it incorporates all of the peripherals we required – USB 2.0, serial graphics, and audio input / output.

The LS-371, like most modern computers, can boot from a variety of data-transport mechanism. In order to simplify the enclosure design and save space, we opted to use the compact flash (CF) memory-slot on the bottom of the board, recognizing that the throughput would likely be slower than other boot device connections. Using a stock LS-371 for testing different boot devices, the primary difference for our particular application was in boot time, as covered more in Section IV applications, once executing, ran at roughly the same speed.

C. Graphics Display

The device includes a 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 more 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" 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 overly that was designed for a similarly sized LCD. A touchscreen controller encodes the overlay's output and sends the encoded data to the SBC via USB. Developing the Linux software drivers for the USB controller we chose presented some additional challenges, which are further discussed in Section IV.

D. 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 performed by the SBC and Ettus USRP requires more power than a similar hardware-based wireless device. 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 customized form factors. These benefits led to our 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 Ettus 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 non-portable operation, an external power supply can be used via a standard 2.1 mm center pin DC jack on the back of the device.

E. 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.

F. 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 resilient to changes in human and capital resource allocation, allowing multiple tasks to proceed in parallel and ultimately saving 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 within the enclosure. 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.

IV. 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 rapid development of applications to demonstrate the capabilities of the prototype devices. This section discusses the choices and implementation of software, issues that arose and how they were resolved, and lessons learned during the integration process.

A. 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 supported the SBC, we found that Ubuntu 6.10 offered the highest level of functionality. After choosing Ubuntu as the host operating system, we had to integrate USB-based touchscreen software and deal with boot issues created by our choice of CF storage.

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 (`http://www.conan.de/touchscreen/evtouch.html`) X.Org event driver, as it was the first solution that compiled with minimal changes – even though at the time it wasn't designed specifically for Ubuntu Linux.

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 boot logs 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.

B. 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. GNU Radio is powerful, scalable, robust software for real-time digital signal processing.

1) Creating Applications: When we began creating applications for the prototype device, the only cross-platform graphical user interface (GUI) available for general use was wxPython ("WX"), a set of Python and C++ bindings for the wxWidgets library. The standard GNU Radio applications use a set of classes that couple WX with GNU Radio, but otherwise do not define a common user interface (UI) – whether command line (CLI) or graphical (GUI).

We found the interface between WX and GNU Radio to be inadequate for efficient application development, and set out to create a more structured framework. The framework we developed ("UI_Block") augments the core software radio functionality of GNU Radio with a powerful yet flexible common UI, enabling the programmer to create more advanced applications more quickly than before.

2) UI_Block Framework: The UI_Block framework provides a full application-programming interface for runtime option selection of both the CLI and GUI. The framework defines specific methods that are overloaded in the application in order to accomplish any required task:

- Add options to CLI parsers, then parse and check CLI options that do not require hardware;
- Initialize any hardware given the current known options, then check any remaining options;
- Initialize each non-GUI block in the signal-processing chain (the "flow-graph");
- Initialize the GUI elements of the flow-graph; and
- Connect all flow-graph blocks and GUI elements, then initialize anything that requires the flow-graph to be connected.

The code for an application written in our framework consists primarily of UI_Block instantiations – including each block's relationship with all blocks connected to it in the flow-graph and GUI. Using separate connections for flow-graph and GUI allows the user to define the ordering of both properties in a single instantiation. Benefits of using this common framework for application development include:

- Programming is class-based and makes extensive use of class inheritance, allowing for maximal code reuse both internal to the framework as well as between applications;
- Default variables and methods, which can be overloaded, provide consistent default CLI flags as well as GUI object placement;
- A bug fixed in any part of the framework is automatically available through class inheritance to any application using the framework;
- Applications using the framework defaults are short, and easy to program and read.

An example of the reduction in complexity achieved by our framework is shown by an application that implements a multi-channel narrowband FM receiver, as depicted by the block diagram in Figure 6. Incoming channels are split using frequency translation and low-pass filters, then decoded in parallel. The decoded audio is summed into one stream, then resampled for audio output. The outgoing channel is a standard PTT, with a twist: logic is built-in to avoid transmitting on any channel that is already in use. The application using just GNU Radio is around 1100 lines of Python script. The application written using our framework is around 400 lines of code.

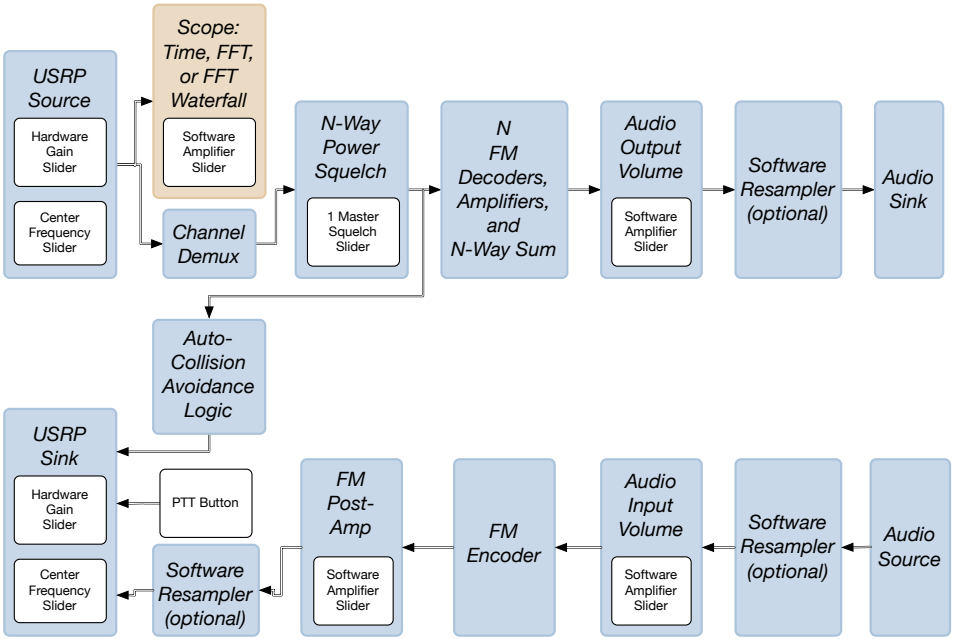

Fig. 6. Block diagram of a narrowband FM multi-channel receiver with push-to-talk transmitter and auto-collision-avoidance logic.

C. 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.

1) Linux Compatibility: "Linux Compatibility" in a generic sense does not necessarily mean compatibility with a specific Linux-distribution. Before picking a distribution, we should have better investigated any additional required software and chosen a distribution that was known to be compatible with the majority of that software.

2) Up-Front Time to Save Time Later: We underestimated the impact of long times 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 more thoroughly investigated other boot methods in order to reduce this waiting time.

3) Prototype Software Installs: Up-front we should have created 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 both install types, we should have obtained storage devices with plenty of extra space since as software progresses it usually increases in the size of both source code and installed files.

4) Revision Control: Keeping track of the various installs of GNU Radio was difficult at best and oftentimes confusing. We should have created 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.

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. We used the source code to determine how to make function calls when help files were not accurate or sufficiently detailed, and to determine 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 the information is made available for searching.

V. Conclusions

Significant progress has been made towards making portable software radios commercially viable, and our efforts contribute to 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 current and future communications problems. Going forward, emerging applications of software radio [9] offer the possibility of revolutionizing the wireless industry through cognitive functionality [10] – allowing radios to dynamically access

spectrum as needed, and moving transmissions elsewhere if legacy radios communicate using the same spectrum.

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.

VI. Acknowledgments

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.

The authors thank Phil McPhee for mechanical engineering design work, Brynnage Design (`http://brynnage.com`) for next-day SLA fabrication of the enclosure, and our Software Radio Group – including Neil Dodson, Andrew Harms, Ben Keller, Marcin Morys, and Yaakov Sloman – for their continuing efforts.

References

[1] GNU Radio Website, 2008. [Online]. Available: `http://www.gnuradio.org`

[2] Ettus Research LLC Website, 2008. [Online]. Available: `http://www.ettus.com`

[3] J. H. Reed, *Software Radio: A Modern Approach to Radio Engineering*, ser. Prentice Hall Communications Engineering and Emerging Technologies Series. Prentice Hall PTR, May 2002.

[4] J. Mitola III, "Software Radios – Survey, Critical Evaluation and Future Directions," *IEEE National Telesystems Conference, NTC-92*, pp. 13/15–13/23, 19-20 May 1992.

[5] R. J. Lackey and D. W. Upmal, "Speakeasy: the Military Software Radio," *IEEE Communications Magazine*, vol. 33, no. 5, pp. 56–61, May 1995.

[6] V. G. Bose, "Design and Implementation of Software Radios Using a General Purpose Processor," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[7] Vanu Website, 2008. [Online]. Available: `http://www.vanu.com`

[8] J. Kumagai, "Winner: Radio Revolutionaries," *IEEE Spectrum Magazine*, January 2007.

[9] J. M. Chapin and W. H. Lehr, "Cognitive Radios for Dynamic Spectrum Access – The Path to Market Success for Dynamic Spectrum Access Technology," *IEEE Communications Magazine*, vol. 45, no. 5, pp. 96–103, May 2007.

[10] S. Haykin, "Cognitive Radio: Brain-Empowered Wireless Communications," *IEEE Journal on Selected Areas in Communications*, vol. 23, no. 2, pp. 201–220, February 2005.

VII. BIOGRAPHIES

Michael L. Dickens is a Ph.D. candidate in Electrical Engineering at the University of Notre Dame. He received a B.S. from MIT in 1991, and a M.S. degree from the University of Notre Dame in 2001, both in Electrical Engineering. He has more than 10 years of industry experience, having worked at the Oak Ridge National Labs (Oak Ridge, TN), Bolt Beranek and Newman ("BBN", Cambridge, MA) including in the Internetworking Division, and most recently the MITRE Corporation (Bedford, MA). His research interests span all aspects of programming for software radios – from operating system boot codes to kernels, signal-processing implementations to user interfaces.

Brian P. Dunn is a Ph.D. candidate in Electrical Engineering at the University of Notre Dame. He received a B.S. degree from Purdue University in 2003 and a M.S. degree from the University of Notre Dame in 2005, both in Electrical Engineering. His research interests lie within wireless communications, spanning from information theory to software radio. He has worked for Crown Audio and BAE Systems, and recently won the 2008 Notre Dame McCloskey Business Plan Competition for a plan to commercialize software radios.

J. Nicholas Laneman is an Associate Professor in Electrical Engineering at Notre Dame and has served as a regular consultant to industry (including two startup companies), government, and intellectual property firms for the past decade. He has a Ph.D. in signal processing and communications from MIT (2002) and his research focuses on software radio, wireless communications, and information theory. He is author or co-author on over 50 publications, many of which are highly downloaded and cited, and co-inventor on 5 US patents, with a number of provisional patents pending. Dr. Laneman received a 2006 Presidential Early-Career Award for Scientists and Engineers (PECASE), a 2006 National Science Foundation (NSF) CAREER Award, a 2003 Oak Ridge Associated Universities (ORAU) Ralph E. Powe Junior Faculty Enhancement Award, and the 2001 MIT EECS Harold L. Hazen Graduate Teaching Award. He is a member of IEEE, ASEE, and Sigma Xi.