University of Paderborn

Computer Networks Group

Evaluating the GNU Software Radio platform for wireless testbeds

Stefan Valentin, Holger von Malm,
Holger Karl

{stefanv|holgervm|holger.karl}@upb.de

February 2006

Technical Report TR-RI-06-273

Technical report

Abstract

Software Defined Radios (SDR) can provide the protocol engineer with wireless testbeds that are fully programmable at the DLC, MAC and PHY. The benefit of this flexibility highly depends on the performance and usability of the specific SDR. In this paper we study the performance and implementational cost of a wireless testbed based on a specific SDR, the GNU Software Radio (GSR) [1]. We implemented a complete transceiver chain [2] which we characterize by implementation complexity. Further, we measure the performance of the testbed running this chain. We found that the GSR platform supports the programmer with a sophisticated SDR programming environment, resulting in low implementational cost. However, this benefit has to be paid for by overhead and some runtime environment idiosyncrasies, resulting in a low system performance.

1 Introduction

Experiments with wireless communication systems can show whether all significant parameters, e.g. related to the scenario or the wireless channel, were taken into account in system design and simulation. Wireless testbeds enable such experimentation.

Besides producing stable and reproducible results, such a testbed should allow changes of the experimental setup. This requires modification of system parameters and adapting functionality to experimental needs. Such adaptation is easily possible for higher protocol layers that are implemented in software, e.g. routing or transport layer protocols. Adapting functions at lower protocol layers that are implemented in or are heavily supported by hardware (such as Physical layer (PHY) and Data Link Control layer (DLC) functions) is nontrivial or impossible with standard equipment. As an example, enhancing 802.11 by an optimized multi hop scheme via changing the Medium Access Control (MAC) or accessing key PHY parameters is an impossible or very time-intense task on standard 802.11 devices. Hence, to support lower-layer wireless protocol engineering, a more flexible testbed platform is necessary. Such a platform is provided by the Software Defined Radio (SDR) approach [3].

SDR, also called Software Radio, refers to an architecture for wireless communication devices where only Radio Frequency (RF) and signal conversion functions are implemented in hardware. All further functions, e.g. modulation, are software implemented, maximizing flexibility and enabling the construction of wireless testbeds that are programmable at the PHY, DLC, and MAC. Such SDR testbeds also support protocol engineers optimizing higher layers of the stack by adapting to PHY and MAC parameters (cross-layer optimization), since here these parameters are easily accessible or controllable.

However, the benefit of this flexibility depends on how the protocol engineer is supported by the underlying SDR platform. The first important criteria here is the performance of the SDR, which hast to suit the targeted scenario. Further, due to the massive increase of software functions in an SDR (compared to a hardware-based radio) a sophisticated programming environment is needed to handle the complex system. Thus, the second important criteria is the programmer's productivity achieved with the programming environment. The resulting development cost and testbed performance are thus highly correlated to the structure and implementation of the specific SDR platform.

In this paper, we evaluate whether a specific SDR implementation, the GNU Software Radio (GSR), is suitable for constructing testbeds and how it supports the protocol engineer according to these two criteria. To characterize the type of wireless testbeds for which GSR is suitable and the development cost of such a testbed, we analyze both SDR performance and overhead of a practical protocol implementation: a full PHY and DLC transceiver chain. The GSR platform was chosen because this open source project combines an up-to-date SDR development environment with freely available and low-cost hardware designs [1].

Although performance is an important parameter, it is not the key aspect for constructing wireless testbeds with SDR. Since in an SDR the programmer can choose which components of the communication systems are included in the testbed, complexity can be limited by including only the functions that are relevant for the experiment. For example, for testing an optimized 802.11 MAC protocol it may not be necessary to include the 802.11 encryption schemes in the testbed. This enables a trade off between the available performance of the underlying SDR system and the needed functionality of the system. Hence, in addition to the

performance aspect we rather focus on usability and productivity of the GSR programming environment and how this affects implementation complexity.

The next section surveys the SDR approach and the GSR platform. In Section 3 the implemented testbed is described. Experimental setup, methodology, and results related to the performance measurements are presented in Section 4. We finally conclude the paper and outline future work.

2 Software defined radio platforms

In this section we introduce the architecture and basic approach of SDR, describe the GSR platform and programming environment, and discuss how this is helpful for a wireless testbed implementation.

2.1 SDR – architecture and programming environments

SDR provides its flexibility by performing modulation and baseband processing in software. The typical architecture is shown in Figure 1. A high-performance Analog-to-Digital Converter (ADC) on the receiver and a Digital-to-Analog Converter (DAC) on the transmitter side act as the gateways between a fixed-function RF front-end and the baseband signal processing functions.

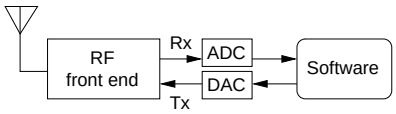

Figure 1: Overview of the SDR architecture

In typical implementations, the converters are supported by specialized digital signal processing hardware, e.g. Digital Signal Processor (DSP) or Field Programmable Gate Array (FPGA), in order to perform digital sample rate decimation (receiver) and interpolation (transmitter). The resulting digitally down-converted data stream, called baseband signal, may be passed to a software-controlled DSP, an FPGA (both suitable for time critical functions in high-performance SDR implementations [4, 5]), or to a general purpose microprocessor. This last approach is used by the so-called *virtual radios* introduced by Bose et al. [6]. On standard PCs, Bose et al. have shown good results combined with easy programmability. The GSR platform, described in the next section, follows this approach and offers a programming environment specialized for SDR development.

2.2 The GNU Software Radio platform

GNU Software Radio (GSR) is an open source project that provides a free software toolkit for developing SDR running on the Linux Operating System (OS) on standard PCs [1].

The programming environment is organized around the principle of constructing a signal processing graph that describes the data flow in the SDR. This graph is executed in an integrated runtime system. The vertices of the graph represent signal processing blocks and

the edges are the data flow between them [7]. Signal processing blocks are functional entities implemented in C++ which operate on infinite data streams flowing from a number of input ports to a number of output ports specified per block. There are primitive signal processing blocks and hierarchical signal processing blocks, which may aggregate both primitive and hierarchical blocks. GSR provides a large and growing software library of individual signal processing routines as well as complete signal processing blocks. Signal graphs can be easily constructed using the object-oriented script language Python.

The runtime system provides dynamic buffer allocation and scheduling according to fixed or dynamic I/O rates of the blocks. The scheduler supports signal graph modifications at runtime. The underlying runtime system can be controlled from Python, meaning that signal graphs can be changed or information can be extracted from the blocks. Furthermore, the environment provides integration of the SDR with the host OS. This enables e.g. to use the SDR as a component of a standard Unix pipeline or utilizing Inter-Process Communication (IPC) for soft real-time signal visualization.

While GSR is hardware-independent, it directly supports hardware front ends via specific source and sink signal blocks. Due to its high integration in GSR and low cost, we focus on the so-called Universal Software Radio Peripheral (USRP) front end designed by Ettus et al. [8]. The USRP consists of one mainboard and up to 2 Rx and 2 Tx daughterboards. While the mainboard performs ADC & DAC conversion, sample rate decimation/interpolation, and interfacing, the daughterboards contain fixed RF front ends or direct interfaces to the mainboard's ADC & DAC. The mainboard is connected via the USB2.0 bus to a PC running the actual GSR. Although the mainboard is able to process signals of up to 64 MHz bandwidth, the data rate of the USB2.0 bus limits the bandwidth of the base band signal to 8 MHz.

3 Testbed implementation

To evaluate this framework's usability and performance, we implemented a wireless testbed on the GSR platform. The testbed contains wireless basic DLC functions and, since only baseband interpolation and conversion is performed at the USRP, all PHY functions on the baseband signal. The full source code of this testbed is available on the project web page [2].

An important design decision is how to distribute this functionality to the different layers of the GSR environment. We decided to implement time-critical PHY and DLC functions (e.g. modulation, framing) in distinct signal blocks; further DLC protocol functions are implemented at the Python layer. Thus, signal blocks can be reused in the sender and receiver graph and the basic DLC functions have full control over the graphs.

3.1 DLC protocol implementation

Automatic Repeat Request (ARQ) and MAC functions are implemented at the Python layer of GSR within the DLC control script. We chose the most simple protocols, i.e., Send-and-wait and ALOHA. Figure 2 shows the resulting DLC frame.

The Python DLC control script includes the protocol and graph control functions for both, transmitter and receiver. If the script is in transmission mode, the protocol parameters frame size N and the CRC result C are exchanged between the Python DLC control script and the signal graph at runtime (Figure 3). Furthermore, the script passes and extracts

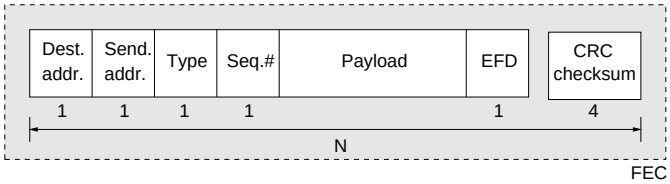

Figure 2: DLC frame layout with CRC and FEC extensions (field lengths in Bytes)

payload to/from the signal graphs and includes functions for graph construction and USRP setup. The total length of the DLC control script is 342 lines, which is, for example, 235 lines less than the ALOHA example in the popular OMNeT++ simulator.

3.2 Transmitter signal graph

The transmitter graph is established and passed to the GSR runtime environment if a frame is passed to the DLC control script. The graph is structured in signal blocks as shown in Figure 3. Its separation was chosen to reflect functional entities of the typical wireless transmission and, as stated above, to enable reuse of the blocks in the transmitter and receiver graphs.

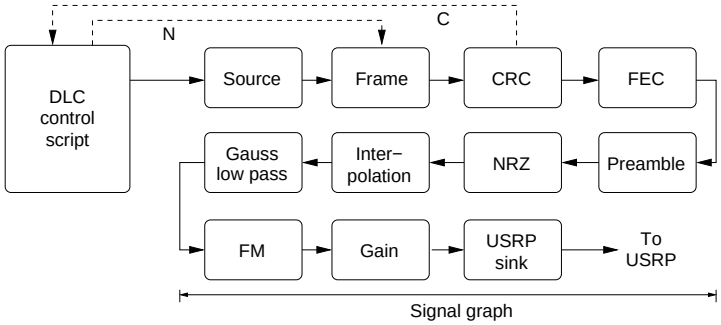

Figure 3: Transmitter signal graph and DLC control script

The transmission path starts at the Source block where a data stream is received from the DLC script. From this payload the Frame block takes $N - 9$ Bytes and constructs the DLC frame. The 32-bit checksum is calculated in the CRC block, which we have implemented using standard library functions [9], and added to the frame resulting in a total frame length of N Bytes (Figure 2). We further implemented an FEC convolution coding block based on Karn's FEC library [10]. After encoding, the frame is extended by the synchronization preamble followed by NRZ channel coding. The resulting digital signal is interpolated and passed to the Gaussian Minimum Shift Keying (GMSK) modulation scheme, which is implemented in the Gauss and Fm blocks. These blocks contain the Gaussian low pass filter and the frequency modulation functions. Finally, the modulated digital signal is multiplied by the gain factor G and passed to the USRP sink block from which it is transmitted via the USB bus to the USRP.

3.3 Receiver signal graph

The receiver graph, illustrated in Figure 4, is started and terminated along with the main script. Once passed to the runtime environment it continuously demodulates the signal from the USRP source block.

The GMSK receiver functions are enclosed in the low pass filter (LP), quadrature demodulator (Demod), and integrate filter (Int) signal blocks. Clock recovery is performed in the Correlator block according to the synchronization preamble. The resulting bit stream is passed to the FEC decoder block, which contains the Viterbi decoder [10]. Then the CRC field is extracted with the help of the EFD escape sequence and tested. If the frame is correctly received the further parts of the frame are extracted in the Frame block.

The implementation of GSR signal blocks is easy [11] and is supported by a large signal processing library. Due to the channel coding, synchronization and modulation functions in this library and due to open source libraries available for CRC and FEC, only 1033 lines of code had to be written for the implementation of the full transmitter and receiver PHY. This is a very small fraction of the PHY implementation in reference [5].

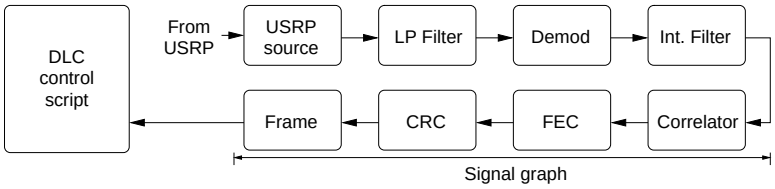

Figure 4: Receiver signal graph and DLC control script

4 Performance study

The performance of the implemented wireless testbed is studied by measurements. We first show the experimental setup, followed by a brief discussion of the chosen metrics and parameters. Finally, we present the measurement results.

4.1 Experimental setup

The experimental setup consists of 2 cross-connected USRP Rev.1 boards, each connected via USB2.0 to one PC. In our case these are 2 identically configured 3.4 GHz Pentium 4 PCs running the standard Linux kernel 2.6.11.4. Each USRP/PC combination and the GSR software version 2.5 running on this PC represents one SDR.

We use unidirectional cable links for cross-connecting the 2 BasicRx with the 2 BasicTx daughterboards (Figure 5). This results in a full-duplex connection via dedicated channels (A, B) for up- and downlink. Although we implemented and measured PHY and DLC functions specific for wireless transmission, e.g. GMSK modulation and the ALOHA MAC protocol, we use a wired connection here. We do this for two reasons: First, transmitter RF front ends are not available so far, they are currently in development [12]. Second, this setup

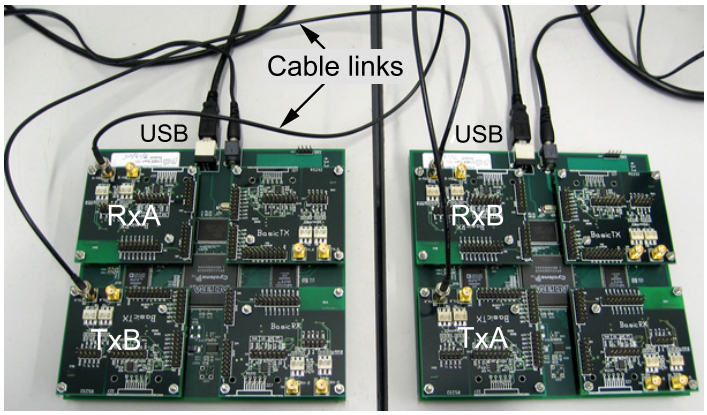

Figure 5: Experimental setup using 2 cross-connected USRP, each board is linked to one Linux PC via the USB2.0 bus

enables us to evaluate the performance of the testbed as such without having to contend with any experimental uncertainties or side effects of the wireless channel.

4.2 Metrics and parameterization

In the given setup, one performance metric is Round Trip Time (RTT). We measure RTT as the time that elapses from the start of sending one DLC frame from one SDR to another to the complete reception of an acknowledgment DLC frame of the same size. Since RTT is measured between the DLC layers of the sender and the receiver the full testbed is included. The transceiver chains of both directions are identical. In such a symmetrical system the RTT comprises *twice* the latencies at the transceiver signal graphs, the time to communicate between PC and USRP and the actual signal propagation delay. The RTT is further studied by measuring the latency, i.e. time between input and output, of each signal block. The results for RTT and block latencies shown in the next subsection are both normalized to time in ms per bit.

In addition to the basic system parameters discussed in Section 2 and 3 we chose the following parameters for the performance measurement: Total length of a DLC frame is chosen to be $N = 1000$ Bytes, FEC convolution coding is performed at a code rate 1/2. For the GMSK modulation scheme we chose a symbol time of 10 μs and 30 kHz for the bandwidth of the Gaussian filter, and $G = 8000$ for the gain factor.

4.3 Results

Figure 6 shows the histogram of the RTT between the DLC layers of the 2 SDR. The mean of the 706 samples is $\overline{\mathrm{RTT}} = 3.14$ ms and the standard deviation is 0.11 ms. From this we can calculate the mean throughput. We assume that no pipelining occurs in the system, which is reasonable since the Send-and-wait ARQ protocol transmits packets sequentially and there is no parallelization in the transceiver themselves. Hence, in this symmetrical system, the mean throughput is $2/\overline{\mathrm{RTT}} = 636$ Bits/s.

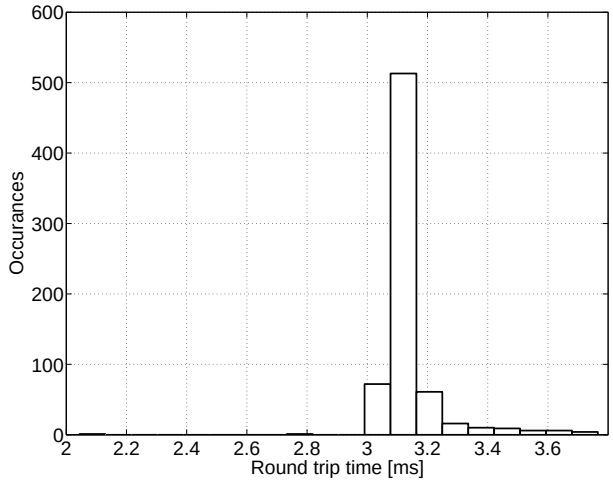

Figure 6: Histogram of the DLC to DLC Round Trip Time (RTT) per bit

The measurement results of the block latencies are shown in Figure 7 and 8. The signal blocks are shown on the x-axis and ordered according to their position in the signal graphs (blocks resulting in zero latency are not included). Both figures show cumulative values (the result for the actual block is always added to the sum of the results for the preceding blocks). The results for the transmitter signal blocks show that most time is consumed by interpolation and modulation functions (Interp., Gauss and Fm blocks). At the last block, the total latency of the transmitter signal graph adds up to $\Delta T_{tx} = 0.42\,\text{ms}$. In the receiver signal graph, filter, demodulation, and correlation are the most time-intensive functions. The block latency sums up to $\Delta T_{rx} = 0.44\,\text{ms}$. Hence, the total delay caused by the latency of all signal blocks for transmitting 2 packets (e.g., data and acknowledgment) is $2 \times (\Delta T_{tx} + \Delta T_{rx}) = 1.72\,\text{ms}$.

Thus, calculating the signal blocks causes 55% of the measured $\overline{\text{RTT}}$. Relative to the total $\overline{\text{RTT}}$ the propagation delay and delays due to USB bus transfer and USRP calculations can be neglected. Thus, the 45% of the RTT not spent calculating signal blocks is overhead caused by the GSR runtime environment and Python control scripts.

The low performance, compared to the symbol time, is caused by two reasons. First, in the DLC implementation the most simple and inefficient ARQ protocol Send-and-wait has been used. This inhibits benefits due to pipelining since only one DLC frame is handled by the transceiver chain at the same time. Allowing even 2 instead of only one outstanding unacknowledged packets would increase efficiency by a factor of about two. Compared to a hardware based system, it bears to mention that the somewhat unusual ratio of low raw data rate vs. high processing overhead in an SDR system need some getting used to. The second reason for the low performance is that the filter blocks of the employed GMSK implementation suffer from a limitation of the GSR framework [13]. Filters in GSR do not produce output unless they are entirely filled. That is, if a filter has M taps, it will not produce data unless there are at least M samples to work with. Due to this reason the filter blocks permanently suffer from buffer underruns which cause constant underutilization of the system.

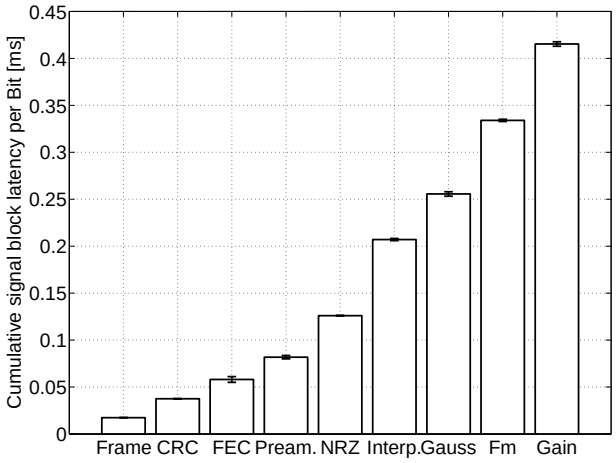

Figure 7: Cumulative average latency per bit for the transmitter signal blocks and confidence intervals at 95% confidence level

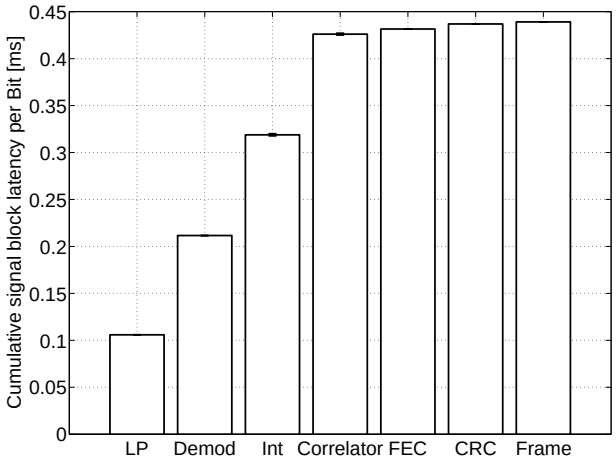

Figure 8: Cumulative average latency per bit for the receiver signal blocks and confidence intervals at 95% confidence level

5 Conclusions and future work

Comparing GSR to other SDR platforms [4, 5, 6] we can conclude the following: First, the performance achieved by the implemented testbed is low. Therefore, the system is not suitable for real time testbeds in the MBit range. Performance is limited by the programming environment and the inefficient filter functions of the GSR signal processing library (Section 4.3). Several projects on this SDR show that throughput is much higher if this problem can be circumvented [14]. However, such circumvention strategies are not immediately applicable to a wireless *protocol* testbed. Currently, they are a better fit with streaming data processing. This issue requires further work. Since GSR is a lively project with an active community, one can speculate that the range of available functions will increase quickly. This community, however, is currently not a typical networking community, as witnessed also by the specific performance problems of communication protocols.

Our second conclusion is, that the testbed programmer is supported by a sophisticated programming environment. We found that this results in low preparation and implementation time and low program complexity. Compared to high-performance SDR testbeds, e.g. [5], the resulting code is short (1375 lines to implement a full PHY & DLC transceiver testbed), modular, and OS and hardware independent. These properties enable to easily share the implemented transceiver blocks with the development community which extends the GSR function library and further reduces development time.

To sum up: The arguments in favor of GSR are its low cost, its large flexibility, its easy-to-use and rich programming environment, and its vibrant community. The main disadvantage is that the programming environment is not perfectly suitable to handling protocols. Therefore, there is a specific price to pay for flexibility when using the GSR in its current form. Whether this price, in terms of performance, for high flexibility and usability of the testbed is reasonable depends on the experimental needs. One possible approach to deal with limited performance is to adapt the testbed complexity to the needed time base. For example, reducing calculation complexity by prolonging symbol time may be an adequate approach if only slow-fading channels are of interest.

This remedy notwithstanding, the essential limitations are the programming and runtime environment's idiosyncrasies. We will look at modifying the scheduler and at increasing the input filter performance. Further we are interested in integrating GSR as a network device in the OS protocol stack, e.g. enabling to pass IP packets to the SDR. With these modifications, GSR should become an even more powerful testbed for wireless protocol prototyping.

References

[1] "GNU Radio – GNU FSF project," Retrieved Feb. 2, 2006 from `http://www.gnu.org/software/gnuradio/`.

[2] Computer Networks Group University of Paderborn, "Project web page: Evaluating the GNU Software Radio platform for wireless testbeds," `http://wwwcs.upb.de/cs/gsr.html`, Feb. 2006.

[3] J. Mitola, "The software radio architecture," *IEEE Communications Magazine*, vol. 33, no. 5, May 1995.

[4] J. Glossner, M. Moudgill, and D. Iancu, "The Sandbridge SDR communications platform," in *Joint IST Workshop on Mobile Future and the Symposium on Trends in Communications (SympoTIC'04)*, Oct. 2004, pp. II–IX.

[5] Signalion GmbH, "SORBAS 101: signalion software radio based protyping system," Retrieved Feb. 2, 2006 from `http://www.signalion.com`, 2005.

[6] V. Bose, M. Ismert, M. Welborn, and J. Guttag, "Virtual radios," *IEEE Journal on selected Areas in Communications*, vol. 17, no. 4, Apr. 1999.

[7] E. Blossom, *Exploring GNU Radio*, Nov. 2004.

[8] D. Shen, *The USRP Board*, Aug. 2005.

[9] D. Walker, "Boost CRC libary," Retrieved Feb. 2, 2006 from `http://www.boost.org/libs/crc`.

[10] P. Karn, "Forward error correcting codes," Retrieved Feb. 2, 2006 from `http://www.ka9q.net/code/fec`.

[11] E. Blossom, *How to Write a Signal Processing Block*, Jan. 2005.

[12] Ettus Research LLC, "USRP sales," Retrieved Feb. 2, 2006 from `http://www.ettus.com`.

[13] J. Lackey, "GMSK python modules for GNU Software Radio," Retrieved Feb. 2, 2006 from `http://noether.uoregon.edu/~jl/gmsk/`.

[14] GNU Radio, "HowtoHdTv," Retrieved Feb. 2, 2006 from `http://comsec.com/wiki?HowtoHdTv`.