

GNU Radio 802.15.4 En- and Decoding

Thomas Schmid
NESL
Department of Electrical Engineering
University of California, Los Angeles
thomas.schmid@ucla.edu

ABSTRACT

The IEEE wireless standard 802.15.4 gets widespread attention because of its adoption in sensor networks, home automation, and other networked systems. The goal of the project is to implement an en- and decoding block for the IEEE 802.15.4 protocol in GNU Radio, an open source solution for software defined radios. This report will give an insight into the working of GNU Radio and some of its hardware components. Additionally, it gives details about the implementation of the en- and decoding blocks. At the end, we will verify the implementation by sending and receiving messages to and from an actual IEEE 802.15.4 radio chip, the CC2420 from ChipCon, and give a small bandwidth comparison of the two solutions.

Categories and Subject Descriptors

C.2.0 [Computer-Communication Networks]: General

General Terms

Software Defined Radio, Cognitive Radio, ZigBee

Keywords

GNU Radio, IEEE 802.15.4

1. INTRODUCTION

A software defined radio is a system which uses software for modulation and demodulation of communication signals. The goal is to use as few hardware as possible. The ideal software defined radio would have an antenna which gets sampled by an ADC and the rest is done in software. Unfortunately, fast ADCs are very difficult to build and very expensive. Therefore, we need some more hardware to first down-convert a band of adequate width from a higher frequency into an intermediate frequency where we then can sample it with our ADC. This is possible today with fairly inexpensive hardware and one can easily process bands of at least 32 MHz.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

UCLA '06 Los Angeles, California USA

Copyright 2006 ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

Software defined radios have been around since the 90s and started out in the analog modem industry, where manufacturers implemented their modems in software, compared as to have the algorithm on a silicon chip. This allowed them to easily upgrade the modulation schemas when new standards came out without changing the hardware. This is an incredible flexibility improvement, though one pays for it with more computing power. One of the major representatives of this group were the so called WinModems which were nothing more than an emulated modem in the main processor of a computer.

From 1992 to 1995, the U.S. Army started a project with the goal of developing a radio capable of operating from 2 MHz up to 2 GHz. The project called SPEAKEasy was successful though there was some dissatisfaction with certain unspecified features. The project went into phase two where these problems were addressed. The goal of the second phase was to make the cryptographic chips faster, such that multiple communications could be handled at the same time. This was not possible with the system in phase one.

The latest U.S. Army project is a joint venture with NATO allies and is called Joint Tactical Radio System (JTRS). It is a multi-billion project with the goal to unify all the available radio communication standards in the different armies into one system. The most interesting aspect of the project is its openness, i.e., most of the documents are available to download from the project website at [1]. The project uses CORBA on POSIX systems to coordinate the modules, and even commercial applications widely accept these concepts.

The goal of this project is less ambitious than the military applications just covered. Nevertheless, it remains challenging to implement a software defined radio solution, even though the processors get faster and faster. In this work, we will present an implementation in GNU Radio [2] to communicate with devices which comply with the wireless standard IEEE 802.15.4 [3]. We will show implementations of a transmitter as well as a receiver module which can be used to send and receive messages from and to such devices.

The remainder of this report is structured as follows. Section 2 gives a short introduction to GNU Radio and its capabilities. The following Section 3 introduces the hardware used in this project, the Universal Software Radio Peripheral (USRP). Next, Section 4 explains some necessary details about the IEEE 802.15.4 standards which are needed to understand this project. This is followed by Section 5 which talks about details of the code implementation and Section 6 which explains how we verified it. Section 7 talks about some problems we ran into during the implementation

and Section 8 presents a summary of future work. Finally, Section 9 gives some concluding remarks.

2. GNU RADIO

GNU Radio [2] is a collection of open source software which when combined with minimal hardware allows to construct radios, and thus turns usual hardware into software problems. The main goal of GNU Radio is to allow easy combination of signal and data processing blocks into powerful modulation, demodulation, or more complex signal processing systems. GNU Radio achieves this by providing simple signal processing primitives written in C++. By using SWIG, an interface compiler which allows an easy integration of C/C++ into scripting languages, GNU Radio provides a simple interface to the signal processing blocks from Python. Thus, the power of scripting languages is used to simply connect the signal processing blocks which can run at native speed without any interpretation.

Several different applications are already written in GNU Radio. For example, there are application which decode HDTV pictures, which can receive and send AM/FM broadcast radio, and there is support for some simple modulation schemas like AM/FM/PSK. Additionally there is an example where they implemented a packet radio system using GMSK modulation and demodulation to transmit packets from one host to an other. The problem with this system is, that GNU Radio doesn't have a good support for packet based processing since it is stream oriented. A DARPA project called "Adaptive Distributed Radio Open-source Intelligent Network (ADROIT)" attempts to change this by implementing a new primitive into GNU Radio, called the "m-block". This will allow a simpler implementation of block based processing and it will also allow to annotate data with meta data. But more to this later.

GNU Radio by itself is not very useful since it needs some hardware to interface to the real world. Fortunately, GNU Radio supports several different hardware platforms [4], like sound cards, and multiple different RF frontends to receive different bands of the RF spectrum. The most commonly used one is the Universal Software Radio Peripheral (USRP) which we will cover in more details in Section 3.

3. THE UNIVERSAL SOFTWARE RADIO PERIPHERAL (USRP)

Matt Ettus developed the USRP [5] as a low-cost and flexible platform for software defined radios. It consists of one motherboard which holds the ADCs, DACs, and a FPGA to do some simple, but bandwidth consuming processing, and to reduce the data rate such that one can push it over the USB 2.0 connection, with which it is hooked up to a PC. The four high-speed ADCs are 12-bit Analog Devices AD9862 and have 64 MS/s. This allows an effective bandwidth of about 32 MHz. The DACs have 14 bit, 128 MS/s and allow to generate signals of up to around 50 MHz. From these figures we can see that the bottle neck is the USB 2.0 connection to the computer which has at most a data rate of 32 MByte/s, thus resulting in a maximum of 8 MS/s of complex signals (16-bit I and 16-bit Q channel).

Depicted on Figure 1 is a block diagram of the USRP receive path. An antenna is connected to either a side A or side B daughter-board. The daughter-boards function is to down-convert a specific band from a higher frequency into

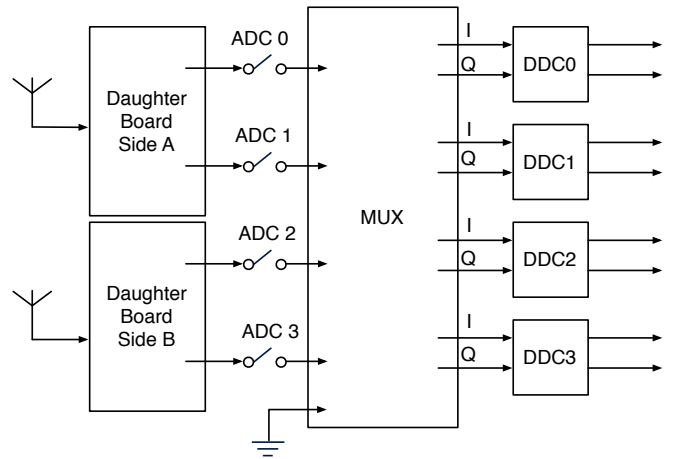


Figure 1: USRP receive path block diagram.

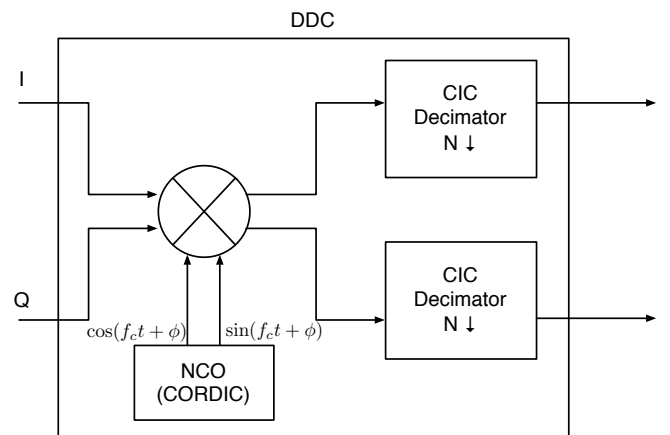


Figure 2: USRP digital down converter.

an intermediate frequency (IF) which can be handled by the ADCs. There are multiple different daughter-boards available for different frequency bands, for example the FLEX2400 for the frequencies from 2.3-2.9 GHz, or the FLEX400 for the frequencies from 400-500 MHz. Note that these two boards cover the two ISM bands which are widely used for different applications like walkie-talkies, WIFI, or IEEE 802.15.4.

The MUX acts as a router and sends the different ADC output signals to the corresponding digital down converters (DDC). It can be configured by software from within python. Figure 2 shows the schematics of the DDC. The two input signals I and Q are multiplied by a sin and a cos at f_c respectively, which yields us the signal centered at 0. This complex signal is then decimated to an acceptable rate such that we can send it over the USB to the PC where it can be processed further in software. Note that the MUX can also configure the Q signal to a constant 0 and thus making the signal real. In some applications this is more than enough and one could achieve a higher sampling rate because we could push more samples per second over the USB.

The transmit path is very similar to the receive path. The samples come from the USB as I-Q baseband into a digital

Symbol	Chip sequence (C0, C1, C2, ... , C31)	uInt32 value
0	11011001110000110101001000101110	3653456430
1	11101101100111000011010100100010	3986437410
2	00101110110110011100001101010010	786023250
3	00100010111011011001110000110101	585997365
4	01010010001011101101100111000011	1378802115
5	00110101001000101110110110011100	891481500
6	11000011010100100010111011011001	3276943065
7	10011100001101010010001011101101	2620728045
8	10001100100101100000011101111011	2358642555
9	10111000110010010110000001110111	3100205175
10	01111011100011001001011000000111	2072811015
11	0111011101110001100100101100000	2008598880
12	00000111011110111000110010010110	125537430
13	01100000011101111011100011001001	1618458825
14	10010110000001110111101110001100	2517072780
15	11001001011000000111011110111000	3378542520

Table 1: The spreading sequence for the 2.5 GHz band of IEEE 802.15.4 for the O-QPSK modulation.

Symbol	Chip sequence (C0, C1, C2, ... , C31)	uInt32 value
0	x1100000011101111010111001101100	1618456172
1	x1001110000001110111101011100110	1309113062
2	x1101100111000000111011110101110	1826650030
3	x1100110110011100000011101111010	1724778362
4	x0101110011011001110000001110111	778887287
5	x1111010111001101100111000000111	2061946375
6	x1110111101011100110110011100000	2007919840
7	x0000111011110101110011011001110	125494990
8	x0011111100010000101000110010011	529027475
9	x0110001111110001000010100011001	838370585
10	x001001100011111000100001010001	320833617
11	x0011001001100011111100010000101	422705285
12	x1010001100100110001111110001000	1368596360
13	x0000101000110010011000111111000	85537272
14	x0001000010100011001001100011111	139563807
15	x1111000100001010001100100110001	2021988657

Table 2: The spreading sequence for the 2.5 GHz band of IEEE 802.15.4 if one uses MSK.

up-converter, where they get interpolated and up-converted to the IF frequency. The MUX then routes the samples to the corresponding DACs before they go out to the daughterboards.

4. IEEE 802.15.4 STANDARD

The IEEE 802.15 Task Group 4 [3] standardized a wireless physical and MAC protocol called IEEE 802.15.4, which targets low-data rate and low-power applications. The standard is freely available from their website and several articles have been published which describe the standard [6], [7], [8] or discuss its safety features [9]. Therefore, I will only give a small introduction to understand the basic concepts which are needed to understand the implementation. For a more detailed description we suggest to read the above cited literature. More specifically, I will only cover the physical layer and some parts of the MAC layer of the 2.4 GHz band which is available world wide. If you need more information about the other bands, please see the cited literature.

Figure 3 illustrates the block diagram for the 2.4 GHz modulation and spreading. First, the pure bits come from the physical protocol data unit (PPDU), which handles the

physical framing, at a data rate of 250 kbit/s. Then, the bits get converted to data symbols of 4 bit, least significant block first, which then get spread according a given spreading sequence (see Table 1). This generates a stream of chips at 2MChip/s, most significant bit first. This stream is then modulated with an offset-quadrature phase shift keying (O-QPSK) modulation with half-sine pulse shape. Figure 4 depicts the modulation of the symbol 0. Note that the I and Q phase are offset by $T_c = 0.5\mu s$. This yields a constant envelop for the complex signal. Also note that the O-QPSK modulation with half-sine pulse shape is equivalent to minimum shift-keying (MSK). One has to be very careful when one uses MSK instead of O-QPSK because the encodings of the different bits differ. J. Notor et al [7] describe a simple schema to convert from MSK to O-QPSK or vice versa. They also mention that the simplest way to convert from one to the other is if one re-encodes the spreading sequence directly as MSK. Table 2 shows the re-encoded values as MSK. Note that the first bit is always set to x . This happens because the value of a bit in MSK encoding/decoding depends on both the I and Q phase bit. Thus, since we don't know the value of the Q-phase for the first bit, we have to set it to x , i.e., unknown.

As shown on Figure 3, the data for the modulation comes from the PPDU. Depicted on Figure 5 is the PPDU frame structure. One layer above the PPDU is the physical layer. It consists of the synchronization header (SHR), the start of frame delimiter (SFD), the frame length and the MAC protocol data unit (MPDU). The SHR itself is four bytes of 0x00. Next, the SFD is defined as the byte 0xA7, and the frame length is the length of the MPDU.

One layer higher than the physical layer is the MAC layer, i.e., the MPDU. It consists of the frame control field (FCF), the data sequence number, the address information, the frame payload, and the frame check sequence (FCS). The explanation of the individual fields of the FCF can be found in the IEEE 802.15.4 standard. In short, the FCF tells the recipient what type of packet he just received and how the address information is stored. Depending on the type of frame, the address can be left out or consist of up to 20 bytes. The FCS is the CRC-CCITT 16-bit checksum of the MPDU. It uses the polynomial $x^{16} + x^{12} + x^5 + x$, which is equivalent to the hex number 0x1021.

5. IMPLEMENTING 802.15.4

5.1 Demodulation

One of the easiest way to demodulate a O-QPSK signal non-coherently is the Low IF Receiver described in [7] which actually implements a MSK demodulator, but since O-QPSK with half-sine pulse shapes and MSK are the same, we can do that. Figure 6 depicts the different blocks involved in the demodulation process. First, the data comes from the USRP into a squelch filter. The squelch filter can be configured to let pass only signals which have a certain dB strength. This avoids unnecessary attempts to decode noise. Additionally, it allows the computer to idle and only has to work when an actual message arrives. In Section 7 we will discuss why we do that.

After the squelch filter, the signal is passed into the FM demodulator which decodes the MSK signal. Then, it is passed into a clock recovery block, which implements a Mueller and Müller discrete-time error-tracking synchronizer. This

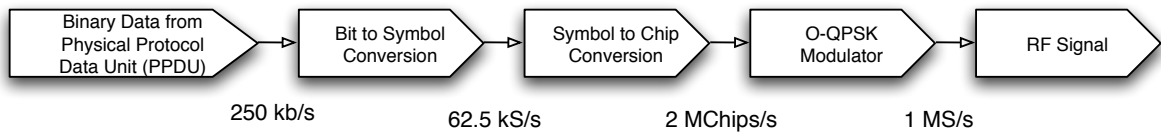


Figure 3: Block diagram for the 2.4 GHz modulation and spreading of IEEE 802.15.4. Additionally, we show the corresponding bit/symbol/chip rates the modules produce.

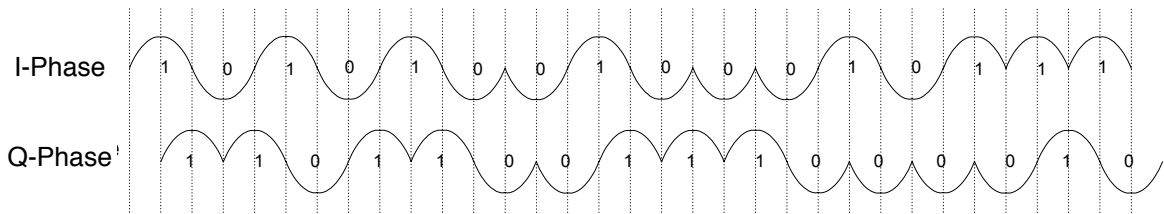


Figure 4: O-QPSK modulation example for the data symbol 0. In O-QPSK, the I and Q phases are shifted by $T_c = 0.5\mu s$.

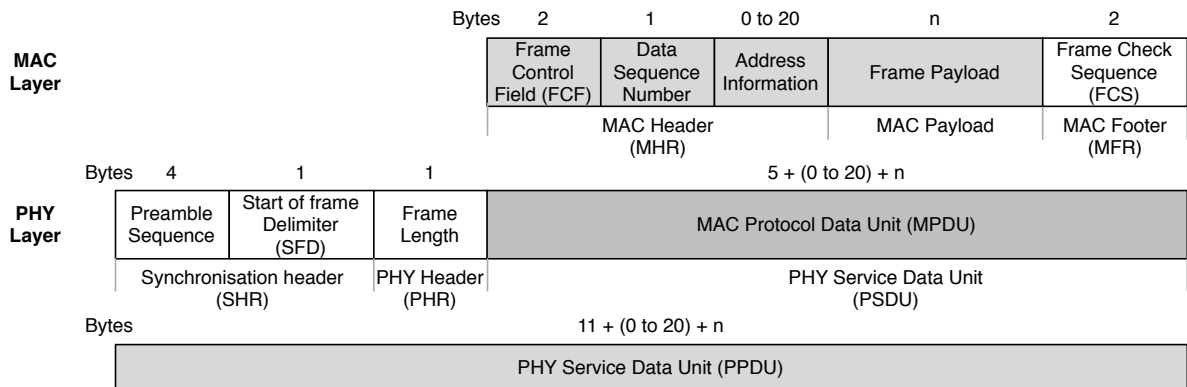


Figure 5: Physical and MAC layer frame structure for IEEE 802.15.4.

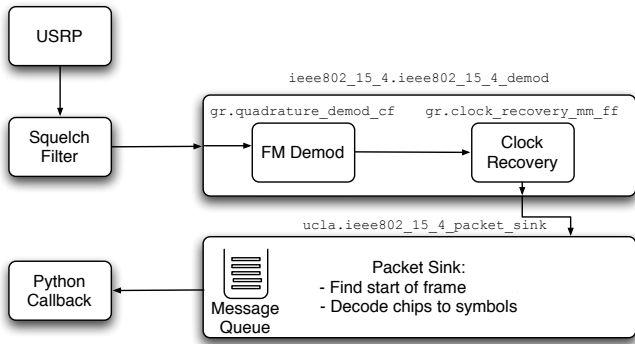


Figure 6: Block schema of the demodulator implemented in GNU Radio.

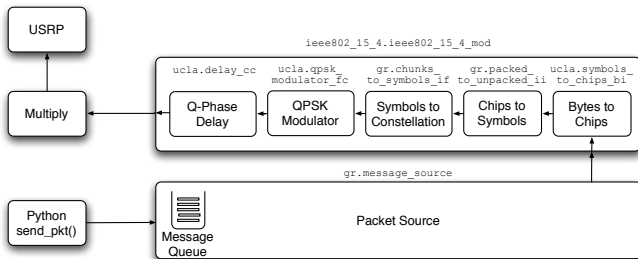


Figure 7: Block schema of the modulator implemented in GNU Radio.

block then outputs the symbols which are ready for slicing. The final stage in the receiving path is the packet sink which is implemented in the C++ object `ucla.ieee802_15_4_packet_sink`. This object implements the physical frame detector and with the help of the packet length field decodes the whole MPDU. Once a complete MPDU is found, it is added to a message queue. In the current implementation, the packet sink doesn't allow any errors in the stream coming from the synchronizer and it needs to find all four 0x00 preamble bytes. This will most likely be changed in the next iteration of the code, where we will allow errors in the decoding process and the block will also detect a frame if it finds less than the 4 synchronization bytes. This will most likely improve the reception rate considerably.

An external python thread is observing the message queue in the packet sink. As soon as there are messages in the queue, the thread starts to call a callback function which then can process the MPDU further. The current test implementation doesn't do anything to the packet except printing it to the console.

5.2 Modulation

As for the receiver, there are multiple possibilities to implement the transmitter. [7] describes two of them. The "2-point $\Delta\Sigma$ PLL Modulator" implements a direct carrier modulation. This is not possible with the USRP since it expects a baseband signal and one can not directly change the carrier output. Therefore, we decided to implement the "O-QPSK Modulator with Half-Sine Shaping". But before we can modulate the data stream, we need to spread it with

the correct spreading sequence. Figure 7 shows the block schema of this process. First, the messages are received from the python application and are put into a queue. From there, the packet source generates a stream of bytes which are sent into the `ieee802_15_4_mod` block. This block first translates the bytes into chips, i.e., it spreads the byte sequence according to the IEEE 802.15.4 protocol. For each byte, it takes first the least significant 4-bit block and spreads it with the 32-bit spreading sequence. Then, it takes the most significant 4-bit block and spreads it again. This sequence of 64 bit is then sent to the next block as two unsigned 32 bit integers. The next block, "Chips to Symbols" translates the integers to a another sequence of integers where each bit in the integer is represented as a 0 or 1 integer, i.e., from one integer input we generate 32 output integers. The block processes the most significant bit of each integer first. Once we have the 0/1 integers, we translate them to the constellation, i.e., we map 0 to -1 and 1 to 1.

The stream of -1 and 1 floats is then fed into the QPSK modulator which outputs the complex baseband QPSK signal with half-sine pulse shaping. To finally generate the O-QPSK signal we pass the complex baseband signal through a Q-Phase delay which delays the Q-Phase by two samples. Before the signal is sent to the USRP, we multiply it by the constant 8000 to scale it to the full range of the 14-bit DAC. At last, the signal is sent to the USRP, where it will be modulated onto the right carrier frequency.

6. IMPLEMENTATION VERIFICATION

We tested the implementation with the Crossbow MicaZ [10] mote which features the CC2420 [11] radio transceiver from Chipcon. On the mote we run SOS [12], an operating system for mote-class wireless devices developed at the Networked and Embedded Systems Lab (NESL) at UCLA. The network stack on the mote is the Chipcon proprietary stack implementation which is compliant with the IEEE 802.15.4 standard.

We created three test scenarios to test the transmitter and the receiver code. First, we programmed a mote to regularly send out a message. This should allow us to find out if the receiver works according to the standard. The messages were very simple and the pure MAC layer payload consisted of 27 bytes. Thus, the total number of bytes sent over the physical channel were 45 bytes. The messages were sent in an interval of 100ms. We sent out five times approximately 1000 messages and calculated the number of messages successfully decoded with GNU Radio. To get a comparison, we also equipped a second mote with a base-station code and recorded how many messages it received. On average, the GNU Radio received 92.8% of the messages the base-station mote received. This is an expected number because the GNU Radio code doesn't allow any errors in the spread sequence. In a second test, we reprogrammed the mote to send messages at 50ms intervals. Here, the GNU Radio code received on average 94.9 % of the messages the base-station received.

The last test was done to check the transmit code of GNU Radio. In this scenario, we sent out the same message the mote produced with the GNU Radio system. A second computer with a second USRP and daughter-board received the messages. Additionally, a base-station mote also received the messages and we counted them as well. The messages were sent out within an interval of 500ms. On average, the

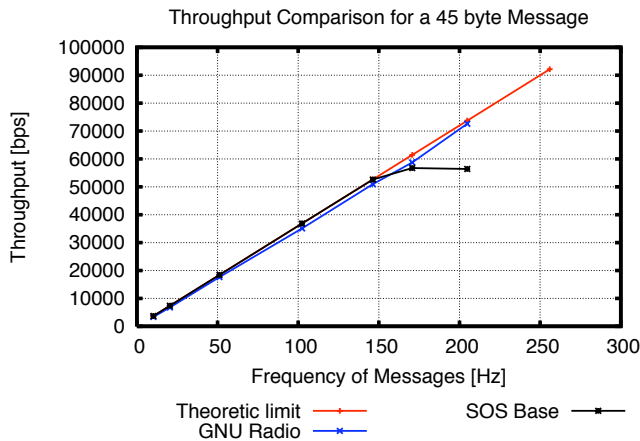


Figure 8: Throughput comparison between the GNU Radio implementation and a SOS base-station using a MicaZ mote.

GNU Radio code successfully decoded 98.6 % of the messages the base-station mote received. This shows that the transmitting code works and that both the transmit and receive code can communicate with other IEEE 802.15.4 compliant transceivers.

7. PROBLEMS AND PITFALLS

In this section we will talk about problems we encountered during the implementation of the physical and MAC layer of IEEE 802.15.4. We hope that it is useful to others which try to attempt similar implementation or want to understand the code in details. It will also explain why we did some special choices of implementation.

The first problem we encountered was the non-coherent receiver. There is a lot of literature on how to demodulate O-QPSK signals, but most of the described receivers are coherent receivers, i.e., the sender and receiver either share the same clock, or the receiver decodes a synchronization signal from the carrier. Unfortunately we don't have direct access to the carrier on the USRP daughter-board itself, and thus we have to use a non-coherent receiver.

Once the demodulation of the O-QPSK signal worked, we found out that the processor we were using, a dual Pentium IV, 2.8GHz with hyper threading and 4 GB of RAM, was not powerful enough to handle a constant stream of data. The USRP experienced buffer overflows all the time if we tried to decode the data in real time. Therefore, we needed to implement a squelch filter in front of the FM demodulator to filter out signals which are too weak. The standard squelch filter in GNU Radio outputs 0 when the incoming signal is too weak. We modified it such that instead of outputting 0s, it doesn't output anything. This successfully solved the problem because the nature of incoming messages is very bursty and it turned out that the buffers of the USRP are big enough to handle the bursts of data. We tried to find out at which point the processor will be too slow to decode the messages coming from a mote. Figure 8 depicts the result of the test. We compared a mote connected through a USB basestation to the computer and the GNU Radio implementation. We can see that at lower rates, the mote

performs better, i.e., it decodes more messages successfully. But once we hit the bottle neck of the UART on the mote (57.6 kbps¹) the GNU Radio receiver keeps it's performance whereas the mote base-station does not. Unfortunately we were not able to go the the theoretic limit of the radio chip, i.e., the full 250 kbps because the mote stopped sending messages above a frequency of 200 Hz.

An other problem lies in the difficulty on when one passes the least significant, and when the most significant block/bit first to the next processing block. Once one figured out that first, the data bytes are chopped into two blocks, where one takes the least significant bit block first, and then we always process the most significant bit first, this problem gets a non issue.

8. FUTURE WORK

There are some smaller problems which need to be investigated and addressed. For example, in the current version the transmitter needs to send an additional 100 bytes at the end of each message or it can not be successfully received by an other device. We suppose that it is a buffer problem and that the message gets cut, i.e., not sent out over the USB if it is too short, though the exact cause is unknown. These 100 bytes hinder us to achieve a constant speed of 250 kbps and we can therefore not achieve nor test this.

Some other improvements will be to implement the higher layers of the IEEE 802.15.4 standard. It might also be possible to implement a full ZigBee stack. The short timing constraints of both standards might be a problem and a new MAC protocol might have to be found which can handle the latencies occurring in GNU Radio.

An other improvement which could be done is to remove some type conversions, i.e., to consolidate some of the blocks. This would certainly improve the speed because less conversion operations would have to be done. Additionally one could try to implement everything as fixed point operations since floating point operations take more time.

9. CONCLUSION

After identifying some initial problems in terms of processing speed of our computer and finding a solution to circumvent it, we successfully implemented an en- and decoding block for the IEEE 802.15.4 wireless standard. We verified the implementation by testing the interoperability with an actual IEEE 802.15.4 radio chip, the CC2420 from Chipcon. The implementation does not implement the whole IEEE 802.15.4 stack since it would be very difficult to achieve the short timing constraints. Nevertheless, the implementation can be used to interact with IEEE 802.15.4 compliant hardware and can be used to further investigate the implications software defined radios have on the MAC and routing layers.

10. REFERENCES

- [1] Joint Tactical Radio System (JTRS). <http://jtrs.army.mil/>.
- [2] Eric Blossom et al. GNU radio. <http://www.gnu.org/software/gnuradio/>.
- [3] IEEE 802.15 wpan2122 task group 4 (tg4). <http://www.ieee802.org/15/pub/TG4.html>.

¹The UART can be configured for higher rates.

- [4] Supported GNU Radio hardware.
<http://comsec.com/wiki?GnuRadioHardware>.
- [5] Matt Ettus. Universal software radio peripheral.
<http://www.ettus.com>.
- [6] E. Callaway, P. Gorday, L. Hester, JA Gutierrez, M. Naeve, B. Heile, and V. Bahl. Home networking with IEEE 802.15. 4: a developing standard for low-rate wireless personal area networks. *Communications Magazine, IEEE*, 40(8):70–77, 2002.
- [7] J. Notor, A. Caviglia, and G. Levy. CMOS RFIC architectures for IEEE 802.15. 4 networks. *Cadence Design Systems, Inc*, 2003.
- [8] J.A. Gutierrez, E. Callaway, et al. *Low-Rate Wireless Personal Area Networks: Enabling Wireless Sensors with IEEE 802. 15. 4*. Institute of Electrical & Electronics Engineers, 2003.
- [9] N. Sastry and D. Wagner. Security considerations for IEEE 802.15. 4 networks. *Proceedings of the 2004 ACM workshop on Wireless security*, pages 32–42, 2004.
- [10] Crossbow Inc. Mpr240, micaz.
<http://www.xbow.com/productsdetails.aspx?sid=101>.
- [11] Chipcon. Cc2420 transceiver.
http://www.chipcon.com/index.cfm?kat_id=2&subkat_id=12&dok_id=115.
- [12] C.C. Han, R.K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. *The Third International Conference on Mobile Systems, Applications, And Services.(2005)*.