Raspberry jAM

ENGR 3420: Analog and Digital Communications Final Project Report

May 10th, 2013

Ian Daniher, Juliana Nazaré and Neal Singer
Franklin W. Olin College of Engineering

1 Introduction

Exploring the field of wireless communications requires hardware in the form of a transmitter and a receiver. For students and researchers with plentiful resources these frequently come conveniently bundled up as little black boxes, where hardware issues are abstracted away via a commercial design and its requisite price tag. But for the individual at home, the high-school, community college or developing world student wishing to dive into comms., access to these black boxes is simply not available, and trying to hack something together frequently results in a poorly operating system that is neither easy to reproduce nor extensibile.

In this project we sought to create an inexpensive, easy to make and easy to use wireless communication link by harnessing the economical yet effective solutions proffered by consumer electronics

Our purpose in attempting this was to provide a system that is adaptable to students and individuals wishing to implement wireless protocols without requiring hardware that is prohibitively expensive or difficult to use. As such our project uses common and inexpensive consumer electronics and is programmed with the Python scripting language.

To determine that our project had the capability to be useful for others, we set performance criteria for ourselves defining minimum range (approx. 150 meters), bit rate (4 kbits per second), and bit error rate (one in one thousand). We believed that if we could meet these requirements during the scope of the time given for the final project in ADComms, then others could surely meet and exceed them with reasonable effort in a reasonable amount of time.

2 System Diagram

A system diagram of our setup is shown in Figure 1. On the transmit side we have a PC controlling a Raspberry Pi via a network socket connection. The Pi transmits through one of its GPIO pins, using a short length of wire as an antenna. This system is outlined in Figure 2 and detailed in the "Transmitting from the Raspberry Pi" section.

On the receive side we have a software-defined radio (SDR) connected via USB to another PC. Using simple software we are able to broadcast a message through the ether and pick it up from some distance away, as explained in the "Receiving Transmissions Using a Software Defined Radio" section later in this paper and detailed in Figure 3.

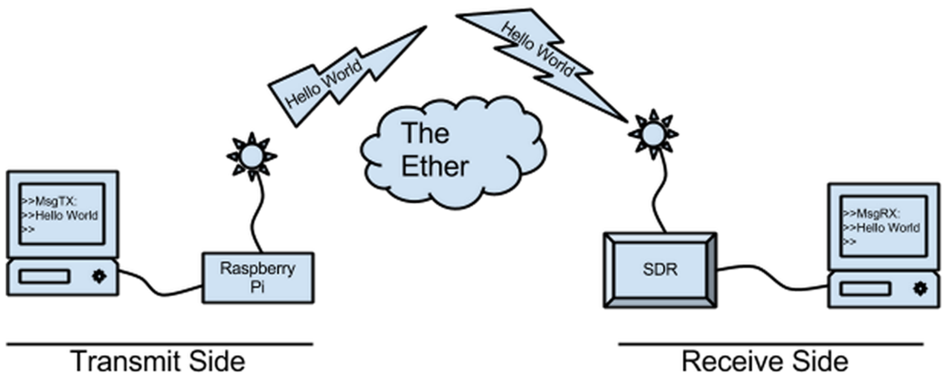

Figure 1: System diagram of the Raspberry(AM)

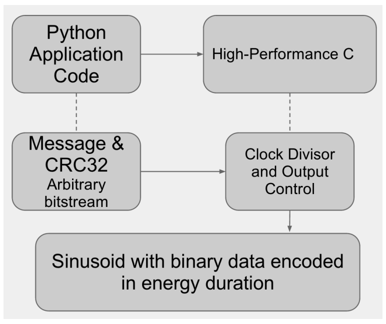
Python Application Code
High-Performance C
Message & CRC32 Arbitrary bitstream
Clock Divisor and Output Control
Sinusoid with binary data encoded in energy duration

Figure 2: System diagram of the transmit side of our system

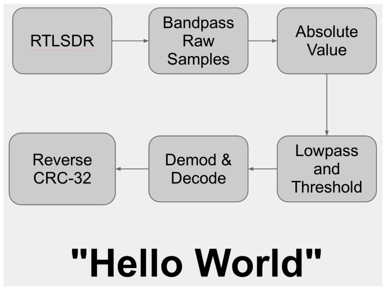

Figure 3: System diagram of the receive side of our system

3 Transmitting from the Raspberry Pi

3.1 Overview

The Raspberry Pi is a cheap single-board linux computer featuring a BCM2835 Broadcomm system-on-a-chip. A full datasheet for this part has not been made available, but a datasheet detailing some of the functionality of this part was released. In it, the clock system for the raspberry pi's physical peripherals is detailed. The 500 MHz base peripheral clock can be divided by a 24 bit fixed point (12 whole / 12 fractional) register and multiplexed directly to a high-current output pin. This gives the Pi the ability to bitbang radio-frequency signals with limited fidelity. Experimental code has been written to utilize the direct-memory-access (DMA) subsystem to modify the divider on the fly, but all attempts to utilize these projects inevitably ended in serious system damage as the kernel and the userland code issued conflicting requests to the direct memory access controller. As a result, our project communicated data using a simple pulse-amplitude modulation scheme where the presence or absence of a chirp allowed us to use the RF channel as a single-level point to point connection. Manchester encoding was utilized to convert a binary message into a sequence of transitions, allowing the message to be demodulated without a prior knowledge of bit composition or duration.

3.2 Implementation

As broadcasting via enabling and disabling the constant broadcast tone required us to rapidly change the output status of a GPIO pin, we needed low-level access to the physical memory of the Pi's CPU in addition to high-precision timing. With this in mind, we elected to build all timing-sensitive transmission abstractions in C and use Python for higher level application code. A hundred lines of C based off of existing projects wass compiled into a shared object file, from which Python has access to the following functions:

- **void usleep2(long us):** High-precision sleep function which accepts a duration in microseconds. Uses nanosleep internally.
- **void setup_io():** General system configuration function; maps memory, sets pin directions, etc.
- **void setup_fm(int divider):** Configures the clock divisor system, accepts a 12.12 fixed point number as an integer.
- **void askHigh():** Enables the output pin.
- **void askLow():** Disables the output pin.
- **void sendByteAsk(unsigned char byte, int sleep):** Iterates through the bits in a passed byte, Manchester encodes, and transmits through sequential calls to askHigh, askLow, and usleep2.

- **void sendStringAsk(char *string, int sleep):** Iterates through the bytes in a passed array, sequentially calls sendByteAsk. This function serves to reduce byte-to-byte latency by reducing the frequency of Python-native code system calls.

After housekeeping functions are called, these abstractions allow the following idiomatic Python function call to broadcast a user-defined message:

```
radio = ctypes.PyDLL("./radio.so")
...housekeeping...
radio.sendStringAsk("hello world", 100)
```

This will broadcast hello world at 5000 bits per second.

3.3 Limitations

The previously described function allowed us to transmit at up to twenty kilobits, albeit with unacceptable bit error rates. The primary failure mode for a transmission is the timing jitter of the underlying nanosleep function. This function, as it runs in the Linux userland, is subject to non-deterministic latencies as the Linux Kernel executes background tasks. For our tests, we ran the underlying Python code with the minimum nice level of -20, giving the best possible operational latency for our sensitive functions.

Writing a lot of the underlying code in Python = **SLOW**

3.4 Possible Explanations

Internet research suggests that the following code block may allow us to disable background kernel operations, decreasing timing jitter, and increasing our effective bitrate, but this has not been tested:

```
struct sched_param sp;
memset(&sp, 0, sizeof(sp));
sp.sched_priority = sched_get_priority_max(SCHED_FIFO);
sched_setscheduler(0, SCHED_FIFO, &sp);
mlockall(MCL_CURRENT | MCL_FUTURE);
```

We may be able to utilize the SOCs on-board high-precision timers to clock a DMA channel to modify either the frequency divider or pin state registers, as discussed above, but all explorations in this area ended poorly.

All code used for transmission can be found at https://github.com/itdaniher/PiAM.
Had we had more time to re-write our code in C, our problems with Python's slowness could be resolved, but for now we believe that our work is a solid proof of concept for an inexpensive software transceiver.

4 Receiving Transmissions Using a Software Defined Radio

To receive our transmitted pulse-amplitude-modulated, Manchester-encoded message, we used an RTL2832U-based USB software defined radio using a R280T highly integrated tuner chip. With a maximum of 2.4 8-bit samples on each of the two quadrature channels. The pyrtlsdr python bindings were used to provide a convenient mechanism for getting raw samples. This library abstracts away automatic gain configuration and normalizes the pair of 8b samples to a complex floating point number between 0 and 1. Convolution with a windowed sinc was used to extract the relevant spectrum. The absolute value of the resulting sinusoid was smoothed by convolution with an array of constant decimals and the resulting signal approximated a pulse train, as shown in Figure 4.

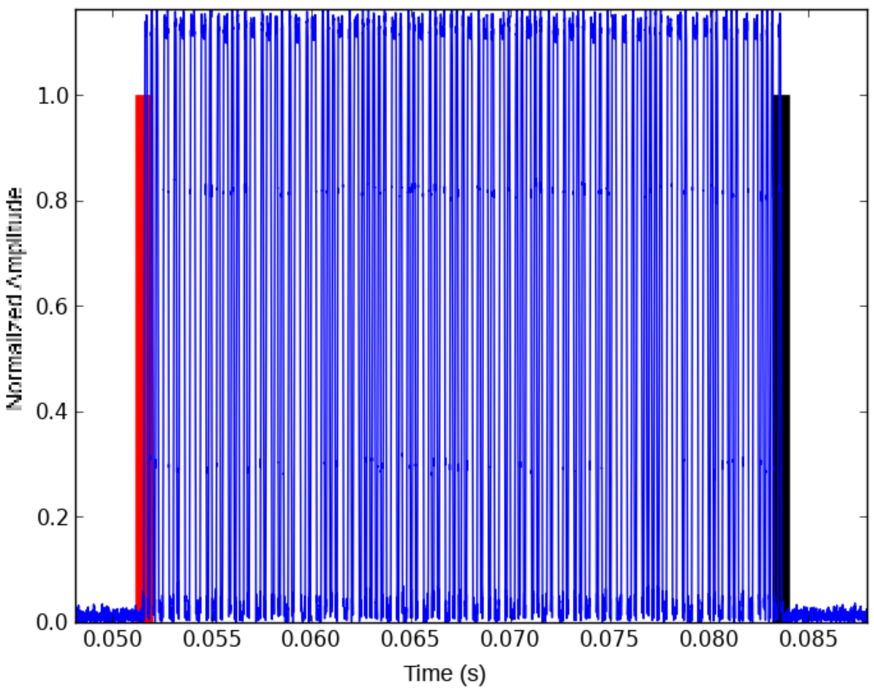

Figure 4: Timeseries with start and stop detection

This pulse train was thresholded and the resulting bitstream was run length encoded, giving a list of pairs of numbers and signal values.

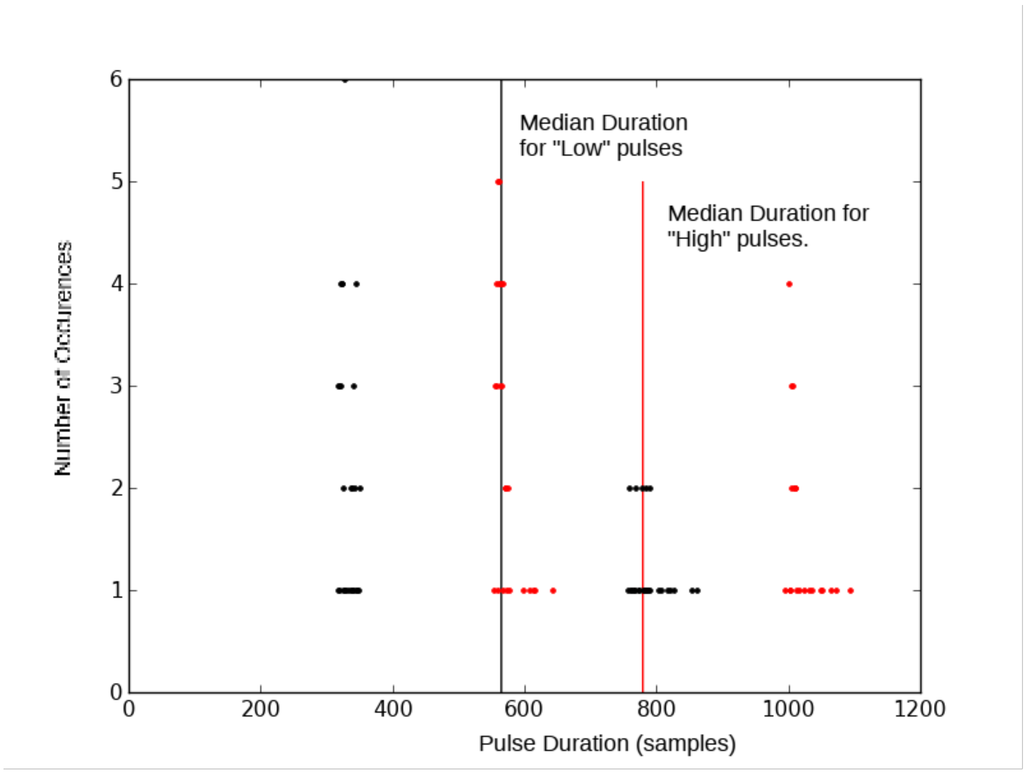

Figure 5: True / False Detection with Bimodal Distribution

These values represented the original Manchester-encoded bitstream fairly precisely, but the nodeterministic latency associated with a userspace transmitter introduced variable delays (see Figure 5). The above image shows the symbol length distribution for high and low pulses. The red marks indicate a high pulse and the black marks indicate a low pulse. The bimodal distribution of long and short pulses corresponds to our understanding of Manchester encoding, but the difference in pulse width for high and low marks provided extensive confusion. Our final script generates two mean values (marked with black and red lines above,) and determines doublewidth marks (corresponding to a zero-one or one-zero transition) to have a duration above the mean and single width marks (repeated binary numbers) to have a duration below the mean. The doublewidth marks were expanded to two consecutive marks of the corresponding high or low value, and the final list of time-normalized values was divided into chunks of two and a binary 1 was recorded for a low / high pair with a binary 0 recorded for a high / low pair.

To implement the previously described signal processing, decoding, and demodulation scheme, the following abstractions were written in Python:

- **getMessageSamples(bittime):** This function accepts a symbol duration (1/baudrate) and triggers a transmission by writing a representation of the the expected symbol duration to a TCP socket on the raspberry pi. This function also encapsulates all function calls made to the receiver and returns a list of 1.2 million complex samples, corresponding to a half second of recorded radio spectrum.

- **clean(samples):** frequency-domain convolution implementation to apply a windowed sinc filter implemented as a 512 tap FIR with 20 kilohertz bandwidth centered around our transmit frequency. This function returns the resulting cleaned sinusoid.
- **demod(messageSamples):** This function processes the sinusoid into a varying-duration pulse train representing the Manchester encoded information. It extracts the expected duration of high and low marks and regularizes double-width marks to two consecutive binary entries in the decoded list. This function returns a list of booleans representing the ideal pulse train of the Manchester encoded message.
- **decode(data):** This function accepts a list of booleans and attempts to build a bit array representing the original method. It attempts to recursively repair framing errors by inserting the the inverse boolean in the case of a pair of booleans being identical, violating Manchester encoding. The returned bitarray can be cast as a list of bytes for printing for as a binary number for CRC checksum analysis.

All receive code is available in rx.py at https://github.com/itdaniher/ADCommsNotebook.

5 Adding in the Cyclic Redundancy Check

We chose to implement a 32-bit cyclic redundancy check (CRC-32) on our system, so as to ensure that we knew when we were receiving a different message than what we were transmitting.

To give a quick overview of how our CRC-32 works. CRC-32 is used as a check to ensure that the messagne that was transmitted is the message recieved. CRC-32 works by appending a 32-bit remainder to the end of the transmission. Upon receiving the transmission, CRC-32 checks to see if the remainder is correct. If it is, it concludes that the transmission was successful and otherwise it throws an error.

To delve more into our CRC-32 function:

1. First, the CRC encode function takes in the bits to be transmitted. In our case, the bits to be transmitted are hello world. We will call these the input value.
2. The function then pads these input bits with n number of zeros. In the case of a CRC-32, the input value will be padded with 32 zeros, which we will call the remainder.

3. Next, the first 33 bits of the input bits are XORed with a 33-bit polynomial divisor (implemented in a bit string). In our case, instead of explicitly defining a polynomial, we pre-computed a lookup table of 256 entries of 32-bit values in order to store the polynomial divisor for each possible byte.

4. After this XOR, the output is assigned to be the new input. This new input is shifted over by one bit, and the next 33 bits are XORed with the polynomial divisor.

5. This pattern of XOR and then shift-by-one continues until the input value XORed with the polynomial divisor carries over to the remainder.

6. This remainder is transmitted along with the initial input value and is received by the receiver.

7. The receiver then checks that the transmitted and received messages are the same by repeating the XOR than shift process, using the same polynomial divisor as the encode function.

8. When the division carries over into the remainder, the algorithm stops. If the remainder contains all zeros, then CRC-32 returns that the message was received correctly. Otherwise, it returns that there was an error.

Using this function, we were able to efficiently tell if we have received the message that was sent. This came in handy as we tested the length over which our system could reliably communicate (explained in the next section).

6 Testing the Distance of Communication

We decided to systematically test how far our system could transmit. To do this, we designed an experiment, where we gradually changed the distance from the transmitter to the receiver, while making sure to keep the two within line-of-sight of each other.

We did this by testing in the AC 4th floor hallway (since it was pouring on the soccer fields for the past two night). We began at one end of the hallway, holding the receiver and transmitter next to each other and ran our transmit and receive scripts. If we saw that the receiver was able to receive hello world, we moved the receiver 10 meters down the AC hallway and tested our system again.

Table 1 displays our raw results for 3 trials. Note: True means that the system successfully received “hello world” with no errors. False indicates that transmitting “hello world” returned errors.

Distance Apart (m)	Trial #1	Trial #2	Trial #3
10m	True	True	True
20m	True	True	True
30m	True	True	True
40m	True	True	True
50m	True	True	True
60m	False	False	False
70m	False	False	False

Table 1: Raw Results

7 Conclusion

We have demonstrated a proof of concept for an inexpensive software-defined transceiver requiring that is both reproducible and extensible, in addition to requiring minimal hardware hacks. While it is not the most sophisticated of transceiver setups, it certainly meets the requirements we set for ourselves, and paves the way for other people to experiment with wireless communications without requiring the resources of an institution.

8 Appendix

The following websites contain code used as a part of building this communications system:

- http://cgit.osmocom.org/rtl-sdr/ contains the source code for the USB driver for the SDR used.
- https://github.com/roger-/pyrtlsdr contains the python bindings for this USB driver.
- https://github.com/itdaniher/ADCommsNotebook/blob/master/rx.py contains the demodulation code. This repository also contains rle.py and fir_coef.py, two small projects used by the demodulation script.
- https://github.com/itdaniher/PiAM contains the transmission code, including the compiled high-performance C library built for doing pulse amplitude modulation.
- Additionally, the Numpy and Scipy numerical computation toolkits were used for implementing DSP primitives such as convolution and the fast fourier transform.