Ardrand: The Arduino as a Hardware Random-Number Generator Status Report

Benedikt Kristinsson
Advisor: Ýmir Vigfússon

December 8, 2011

Introduction

It has been somewhat of an urban legend surrounding the Arduino that it can easily be used as a hardware random number generator. The idea for this project came while working on the TSeense project[1] in the summer of 2010.

The Arduino has 6 analog inputs (see `http://arduino.cc/en/Reference/AnalogRead`) that are 10-bit analog to digital converters.

When not connected to anything these outputs should be influenced by analog noise, much in the same was a static on a radio that is not tuned in to anything. This is dependent on the design of the connectors and some other factors.

Goals

The original, yet optimistic, goal was to design and implement a True Random-Number Generator on the Arduino. As the project has progressed it has become evidently clear that it is not possible to do this properly (without adding some hardware, such as a diode and read the noise in its p-n junction).

However, Arduino themselves claim in the Arduino Reference Manual[2] that `analogRead()` can be used to generate a "fairly random" integer to seed the PRNG algorithm[3]. I claim that the vanilla output of `analogRead()` is not even "fairly random". The goal of the project is now to show this and ultimately write a program that given a PRNG sequence from the Arduino, finds the seed value.

Attempts at harvesting entropy

The Arduino toolkit has the `analogRead()` function that reads from a given analog pin on the board and returns a 10-bit integer. This is what we are trying

[1] `http://code.google.com/p/tsense`
[2] `http://arduino.cc/en/Reference/HomePage`
[3] `http://arduino.cc/en/Reference/Random`

to use in order to extract entropy on the Arduino. The function maps voltages between 5 and 0 volts to integers in the range [0..1023].

The raw readings should be influenced by analog noise but our experiments have shown that the output is fairly regular, although operating conditions can influence it greatly. We see a much wider range of values when operating in either heat or cold (such as a heating element or freezer). As Figures 3 and 4 show, there are certain interference patterns that show up (they are rather interesting). Although we are not sure what causes them, electrical fluctuations seem like a worthy candidate. In the case of the fridge, it stops "operating" when it reaches a certain temperature and starts up again when the temp rises too high. It seems very likely that it gives away electromagnetical waves that "disturb" the readings of `analogRead()`. These patterns might also be a product of the nature of the analog pins themselves, or their manufacturing process. Unfortunately we bricked one Arduino by leaving it in the fridge overnight.

As we can see in Figure 1, the output of `analogRead()` spans a very short range. The different locations even give off very similar readings at the same or similar temperatore. Notice that the purple values are slightly different — they are collected inside a standard Dragon computer case. The temperature is slightly above room temperature and it might be affected by electromagnectic waves amongst other things.

Figure 2 shows a short span of a sample collected at room temperature in a bedroom. Just by staring at the plot for a minute or two, we can see that there isn't much going on — the entropy is very low. This causes very low performance in generating the random bits. We have seen bitrates between 3 bps and 25 bps.

This data should disprove the claims put forth in the Arduino reference manual.

Requires further investigation

Note the drop at the beginning for all readings. This needs to be investigated further. We have collected some data but nothing conclusive yet.

The (in)feasibility of the Arduino as a TRNG

So far we have implemented and tested roughly three different algorithms to generate random bits. So far none has even passed the monobit test according to the FIPS specifications[4].

RAND Algorithms

Two algorithms have been implemented and tested. One more has been implemented but not tested enough (due to the unfortunately-timed-bricked Arduino).

The `Mean-RAND` algorithm is implemented by keeping a list of the k last values and their mean. Then we compare the new reading to the mean and evalute to 0 if it is less, otherwise 1. To remove bias and lessen correlation we run it through the von Neumann-box.

[4]See Menezes Ch.5 and FIPS 140-1

The von Neumann box is a function that inputs two bits. If we pass 00 or 11 to it, it discards the bits. If we input 01 then it outputs a 0 and if 10 then it outputs a 1.

Listing 1: The `Mean-RAND` algorithm in Python-ish pseudocode

```
def meanrand(n):
  buf = deque([0]*k)
  meanval = sum(buf)/len(buf)

  for i in [0..n]:
    while True:
      meanval -= buf.pop()/k
      buf.push(analogRead())
      meanval += buf[-1]/k
      m = ceil(meanval)

      b0 and b1 = 1 if analogRead() > m else 0
      if b0 == b1:
        discard
      else:
        break      # Break out of the vN-box
    if b0 == 1:
      yield '1'
    else:
      yield '0'
```

The `Updown-RAND` algorithm first reads an initial value v_0 which is then used to determins if the next bit value v_1 is 1 if $v_1 > v_0$ and 0 otherwise. We do this twice, i.e. we collect $v_{1,0}$ and $v_{1,1}$ and compare them with the von Neumann box until we obtain a legit bit.

Listing 2: The `Updown-RAND` algorithm

```
def updownrand(n):
  v0 = analogRead()
  for i in [0..n]:
    while True:
      v10 and v11 = 1 if analogRead() > v0 else 0
      if v10 == v11:
        discard
      else:
        break    # Break out of the vN-box
    if v10 == 1:
      yield '1'
    else:
      yield '0'
```

A third algorithm, `MixUpDownMean-RAND` has been implemented but we do not have any conclusive results on it yet, since the Arduino meant for testing it is bricked. It works by generating one bit b_M from `Mean-RAND` and another bit b_U from `Updown-RAND` and then return $b = b_M \oplus b_U$

Listing 3: The `MixUpDownMean-RAND` algorithm

```
def mixrand(n):
  m = meanrand(n)
```

```
u = updownrand(n)
for i in [0..n]:
  while True:
    bm = m.next()
    bu = u.next()
    if bm == bu:
      discard
    else:
      break       # Break out of the vN-box
  yield bm^bu
```

Testing for entropy

The Monobit test measures if the number of 1's and 0's are approximately the same, which we would expect from a random bit sequence. We let n_1 and n_o denote the number of 1's and 0's, where $n = n_0 + n_1$. Then the stastic used is

$$X_1 = \frac{(n_0 - n_1)^2}{n}.$$

The FIPS 140-1 docmument specifies certain requirements that a RNG has to satisfy. The odd thing is that FIPS doesnt say anything about the statistic X_1 itself for the Monobit test. Rather, it says that a bitstring s of length 20,000 has to to have a value n_1 that satisfies

$$9654 < n_1 < 10346.$$

Neither the `Mean-RAND` or `Updown-RAND` have been able to produce a satisfying n_1. Further, due to the lack of entropy, generating 20,000 bits has taken anywhere between 13 minutes to 108 minutes in our experiments (Or, 3 bps to 25 bps).

Algorithm	n_1 (best observed case)
`Mean-RAND`	Fill me in
`Updown-RAND`	Fill me in too

Until we have something that actually passes the Monobit test (the most basic of the tests) we will not use more advanced tests.

Disproving claims made by Arduino

Since we have found that using a vanilla Arduino as a RNG is infeasible, this changes the goal of the project. Instead we will show that what the Arduino manual claims to be a "fairly random" seed is, on the contrarly, not that random at all and a very bad seed source.

First off, `analogRead()` returns a 10-bit integer. The `randomSeed()` function inputs a regular 32-bit integeger, i.e. there are 22 bits left "unutilized". Instead of a full range from $[0..2^{32}]$ we have the much smaller range $[0..2^{10}]$.

By looking at Figure 1, we can see that for almost every single reading under normal operating conditions we read a value roughly in the range [210..375]. This means that a possible adversary would only have to try roughly 160 different seeds or psuedo-random sequences. With a truly random seed, he would

have to try 2^{32} different seeds, but in the case of `analogRead()` he can do much better.

Even under the more extreme conditions there is a fairly low number of seed values or sequences the adversary would have to try. In the case of the freezer, we see output in the range of [50..410], that is 360 possible values. The most interesting case is the heating element, were the large cluster of values is in the interval [200..500] but we see values all the way down to 0 much more commonly than in the case of the room temperature experiments.

The drops in the beginning have to be investigated further before we can say anything conclusive about those values. The bottom values appear rather linerally and non-randomly and seem to happen when we read with short enough intervals and the serial buffer fills up or sends malformed data (We have found out heuristically (?) that 0.02% of all readings are either empty of malformed strings).

Fallback

The original goal obviously doesn't hold any more and we have shown that using `analogRead()` as a seed source is a bad idea. The fallback-plan is to implement a program that finds the seed given a PRNG-sequce.

Figures

Will be small in print, buf if you are reading the PDF you can zoom in (.png pictures)

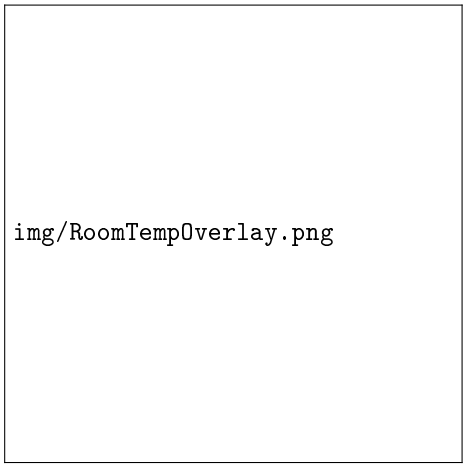

Figure 1: Readings of `analogRead()` at room temperature

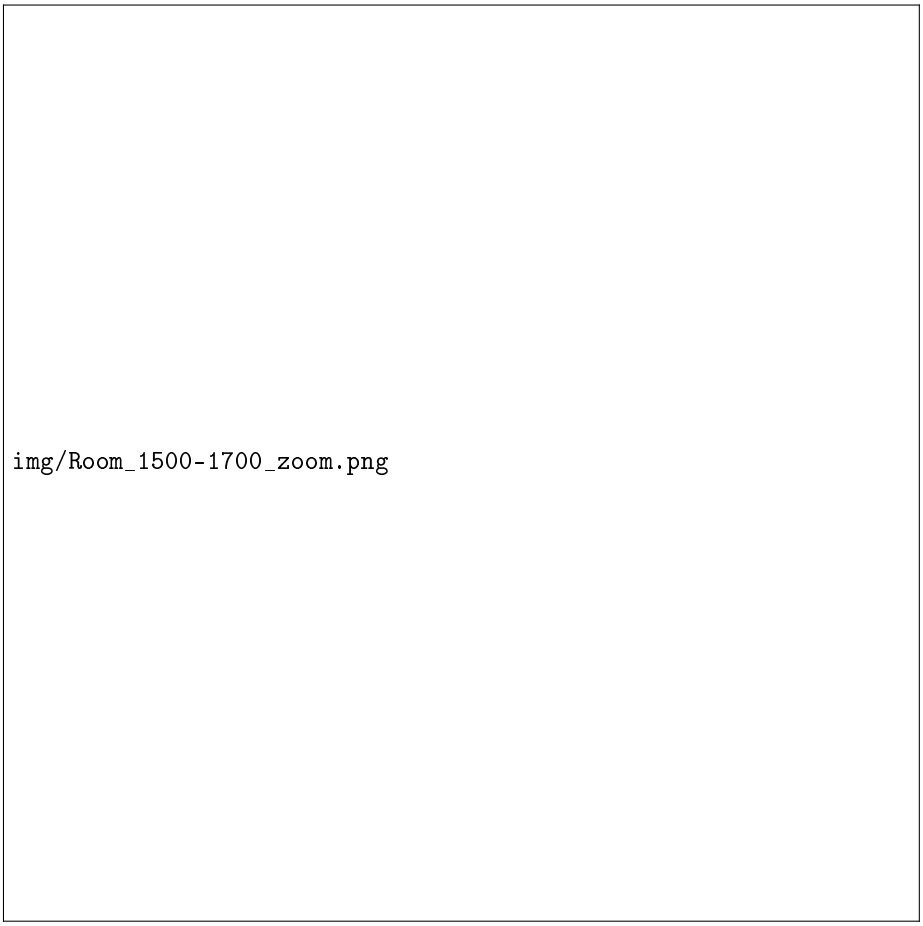

Figure 2: Reading at room temp. Shows $x = [1500, 1700]$

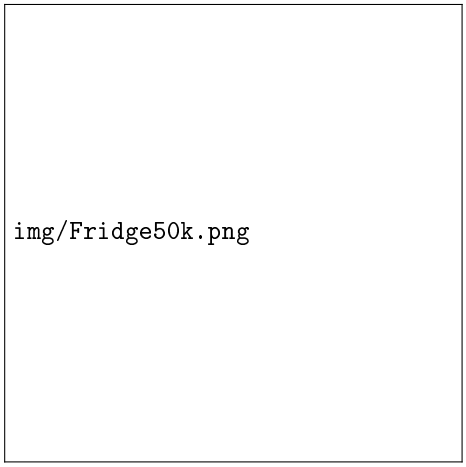

Figure 3: Readings inside a fridge at 1°C

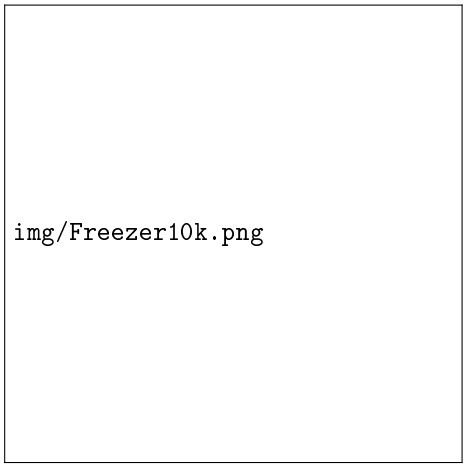

Figure 4: Readings inside a freezer at $-11°\mathrm{C}$

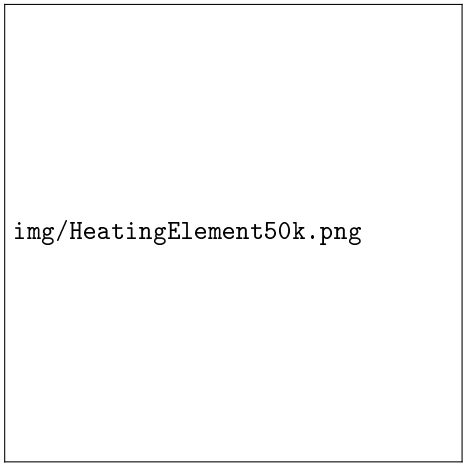

Figure 5: Readings on a heating element at approx $40°\mathrm{C}$