Random Bit Generator for cryptographic strength random numbers

Randomness is one of the great mathematical and philosophical challenges of our times, up there with infinity and quanta. If you are interested/curious about randomness and its properties, check out this site https://www.random.org/randomness/.

One assured way of generating randomness is by sampling natural phenomena - CloudFlare, an internet services provider, actually uses a bank of lava_lamps. Images are captured and processed, and the position of the lava is used to provide a seed and salt for their random number generation. One day, we may finally understand the quantum universe only to discover nothing is indeterminable, but for now we'll assume it is.

From a security point of view, really strong encryption needs a random element added to hashes called SALTing. The salt should never be re-used or fixed, so ideally you use a random generator to create a new salt each time.

The problem with TRUE random is it is actually very difficult to achieve and programmers often resort to methods like setting the seed for their pet RNG to the time or similar. This might seem to be a good approach but most RND() functions use a logic algorithm to generate what seems to be random but are actually part of a long sequence of a few million numbers. Modern computers can storm through them to find your starting point in the sequence in seconds and then your salt is useless. If you are stuck with such a pseudo-random generator (PRNG), try to seed it with something as ransom as possible, un-guessable (non-deterministic). Sampling the analogue value on a floating pin, the bytes throughput on a network connection etc. is quite good and such a precaution can bring a PRNG close to the realms of True random generators (TRNG).

You can prove all this to yourself; in your favourite language, set the random seed to a fixed value then use the RND() function to obtain a random value. Repeat. The RND() value is the same each time - you are seeing the sequence here at work and once your start point is determined, the following numbers are entirely predictable.

```
'VB Random test
    Randomize 50000
    Debug.Print Rnd*100000
    Randomize 50000
    Debug.Print Rnd*100000
    Randomize 50000
    Debug.Print Rnd*100000
```

Randomizing on a timer will produce a different result each time (likely) but because each RND() number is from a sequence, it is crack-able - no matter where you start in it. RANDOMIZE only chooses the start point. As an aside here, it is entirely plausible that a seeded PRNG might want to generate the same numbers each time - don't assume that “random” has to be true random for every application.

Modern languages on servers and PCs do have methods to generate cryptographically strong numbers and processors are now appearing that have true random generators fabbed onto the chip (probably using a solution similar to that described below). There is an application in the world of low-end microcontrollers too - hence this article. PIC32MZ microcontrollers do have a true RNG on the chip and MMBasic for MMX uses this for its RND() function - and correspondingly has no RANDOMIZE

statement (because there is no “list” in which to choose your starting point). Not much point in you reading further if you are working with an MMX!

There is a really good article on salting here: https://crackstation.net/hashing_security.htm with rights and wrongs.

Usually, to generate a random number, you sample a number of bits and gradually left-shift the result into a variable - one bit at a time. If you want a simple number that will probably be good enough but to support RND() of high-level languages, you need to do a bit more processing. RND() is a fractional value in the range 0⇐n<1 - so it can be zero but cannot be 1 - any decimal number in between is fine. The algorithm for converting your random number to RND() is as follows:

- Get n bits as your random number and call it t
- Divide 1 by 2^n
- multiply by t

so in a basic language it might be

```
n=16           ' get 16 bits
v=GetRandom(n) ' v is the returned random value
t=1/(2^n)*v    ' true random is now in the range 0<=t<1
```

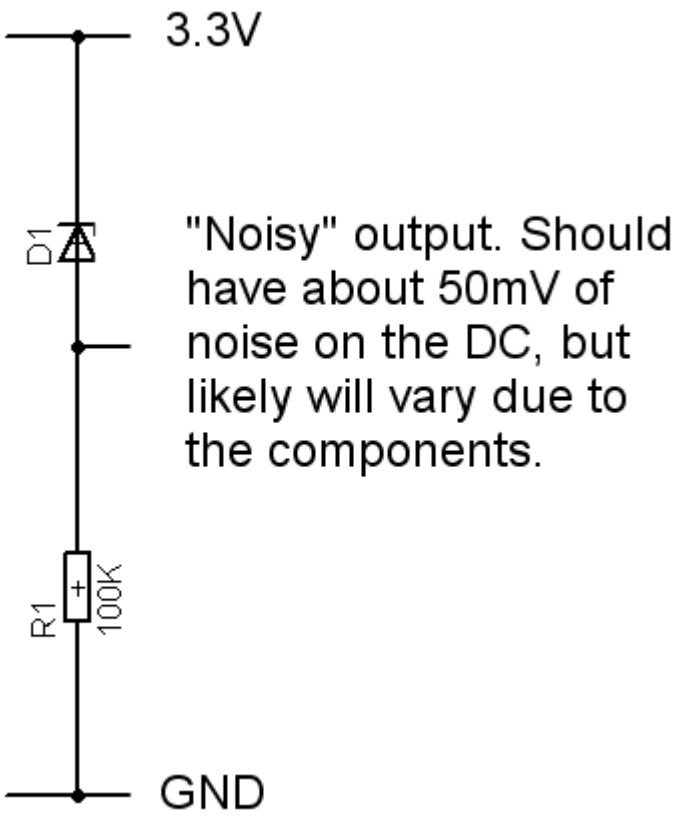
to Illustrate, let's assume 16 bits; 2^16=65536. However the maximum value that can be held in 16 bits is 65535 So, 1/65536=0.0000152588 multiplied by 65535 is 0.999985 Thus we have satisfied the requirement that RND() can never quite reach 1 and is now in a fit state to be multiplied by any factor to place the random number in whatever range you choose in the usual manner. Obviously the more bits you return, the greater the granularity but also the slower the function. Choose a number that works for your application - if you only need a yes or no answer then fetching 1 bit will work fine and RND() will either be 0 or 0.5

The circuit below generates a continuous stream of bits (1 or 0) from a white noise generator and you simply sample as many bits as you need. So if you want a 16 bit random, you sample it 16 times… This makes it really simple to interface to a microcontroller requiring just a single input pin and some software. The noise is truly random - it comes from the (as yet) indeterminable quantum effects of electrons falling through a silicon PN junction. The different arrival times on the other side (and associated rapid change in electrical potential) produce spikes which vary according to the number of electrons falling through at any one time. This is a form of Shot Noise and there is no pattern. Here is example code; True Random Generator (companion code for circuit). Diligent trimming will be required to avoid a noise source that is biased one way or the other - it is really difficult to get proper random :o/ There are computational methods to remove bias, Software Whitening is one but it can be intensive and require much re-sampling with very badly biased input.

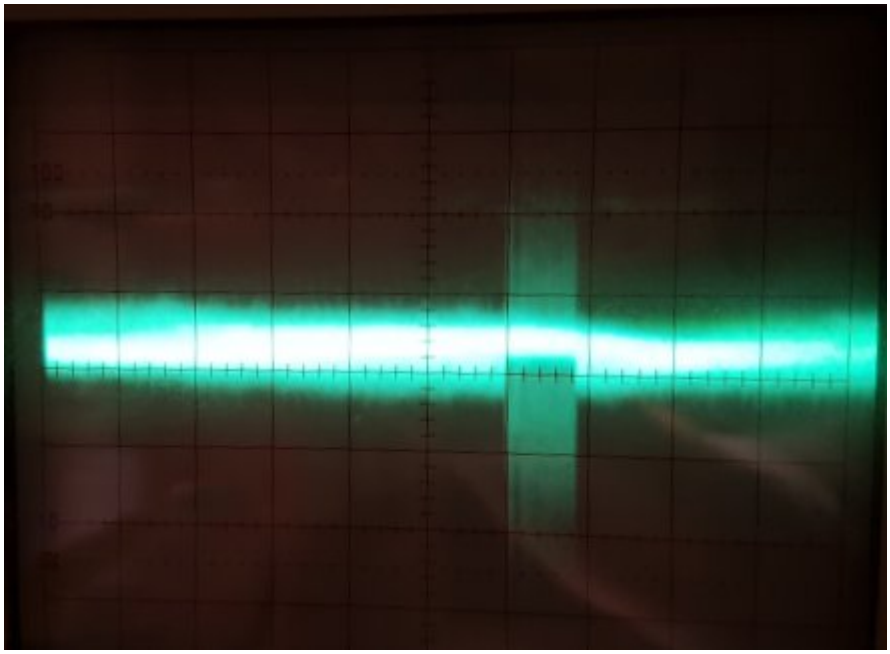
There are loads of white noise generator circuits on the web and you don't have to use what I did. There are also some nice digital solutions but they tend not to be true random, again just a sequence. Have a look at this one http://electricdruid.net/white_noise_source which you could use to sample directly and avoid all the analogue stuff below. It is good for 40,000 samples/sec for 142 million years before the sequence repeats! Not true random but big enough that it would be very difficult to find the start in a sequence of 179 million, trillion (179,124,480,000,000,000,000) bits. The start point is not settable so any crack attempt would still have the advantage of knowing the sequence started sometime in, say, the last five years (since last powered-on) which; cryptographically speaking, shortens the odds.

The circuit is quite straightforward and relies on a zener diode in “avalanche” - i.e. the PN junction has been forced to conduct “backwards” and so broken down. Zeners are a form of diode that utilise this phenomenon to create very specific points at which this breakdown occurs, so they can be used for voltage references etc. Due to the breakdown, a Zener diode is a noisy little beast while in avalanche. By siphoning off this noise through a capacitor and amplifying it (a lot) we get a pretty strong noise signal.

So you need a good noise source - ordinarily this circuit would be typical.

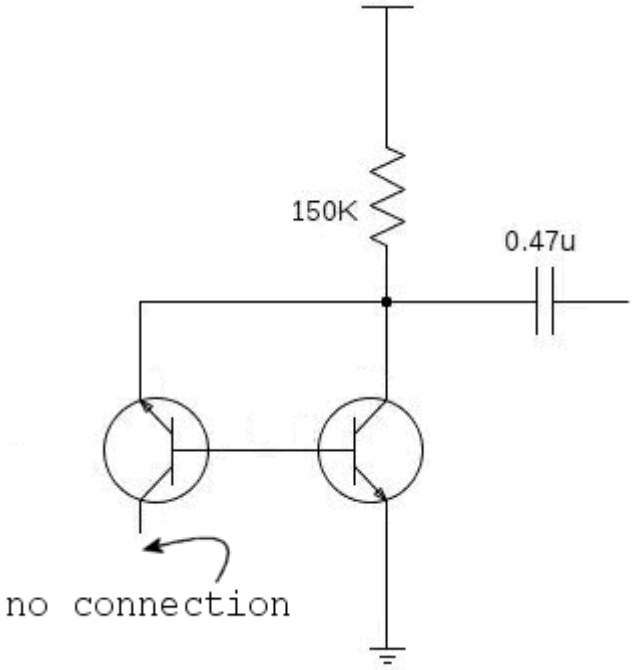


…but while playing with it (and it was noisy with about 1MHz hoot - which is fine) a strange noise pulse-train was occurring at about 40mS intervals. I tried to isolate it but with no luck (yet). I suspect the zener I was using wasn't happy at the low voltages and was “surging” in some manner.

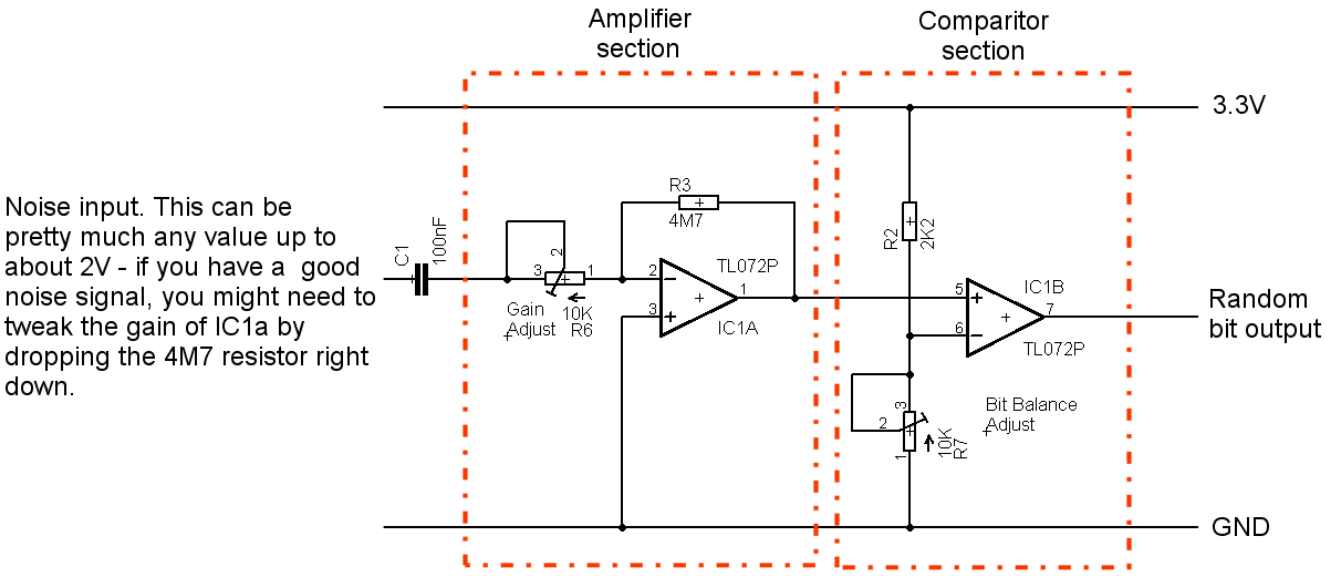


You can see the main trace has a lot of noise at about 60mV but also a band of hi-amplitude noise way off the scale - you can also see this band is a composite of lots of spikes - which would be perfect if only the entire signal was made up of these. The problem is, this signal is no good because the main signal has to be amplified so we can just see its peaks, but then along comes this big splurge which will saturate the amplifier and produce a long train of 1's from the comparator which will unfairly bias the random… so not random at all, a big train of ones at 40mS intervals - Great. :o(Usually I spend a lot of time getting rid of noise, now I have noise on my noise… A way around this would be to bias the noise signal to 50% VCC - then the comparator (also set to centre on 50% VCC) wont' care too much about the signal only which side of the threshold it is. This way the noise signal can be as, err, noisy as it likes.

Here is another noise generator (found via google search), this time using the base-emitter junction of a NPN transistor by forcing it into avalanche - pretty much any small signal transistor will do, 2N3904, 2N4124, BC107 etc… This is likely to be destructive over time and you may find that using the transistor in the “wrong” way like this means you'll have to repair the noise generator in the future (Zeners are manufactured so as not to be damaged over time in this method of use). The second transistor is a pre-amplifier to bring the noise to levels that can be used in your main amplifier stage.



Assuming then you have a good solid noise signal, it is fed (after being amplified) into a comparator which toggles its output depending on whether the input is above or below a threshold. This is then input to your microcontroller. Normally circuit designs go to a lot of trouble to eliminate noise, here we actively seek it. Here is the business end of the random bit generation.



The trimmers can be set to give the amplification of the noise, and then to set the threshold at which the output toggles. Using an oscilloscope, monitor the output of the first OpAmp to get your noise signal as big as you can without it clipping (flat bits at the top of the waveform) and try to trim for even signal distribution. Write a small prog for your microcontroller to continually monitor the input pin and print either 1 or 0 depending on the input. Adjust the second trimmer so the stream of 1's and 0's doesn't seem to be overly biased towards one or the other and that is it. This will also demonstrate if your noise frequency is too low (lots of groups of digits, seldom on their own).