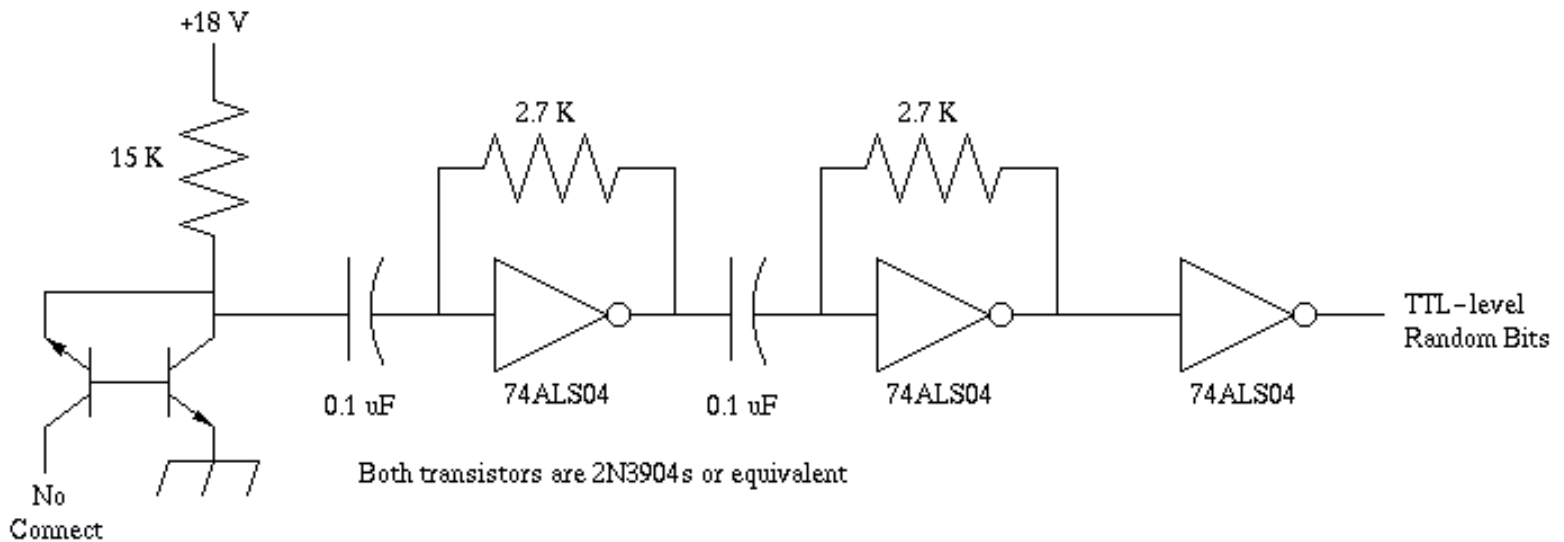


Hardware Random Bit Generator

The article appearing below was one of my recent posts to the [sci.crypt](#) newsgroup. These are some notes about building a hardware random bit generator. Johnny von Neumann once said that anybody who contemplates arithmetic methods for the generation of random numbers is in a state of sin. What he meant by that is this: if you try to generate random numbers using only software, the results will necessarily depend on the state of your machine at the outset of the process, and are therefore (at least in principle) insecure. That is, somebody could conceivably reconstruct your *allegedly random* bitstream.

Now in fact, programs like PGP and Linux's /dev/random driver do a remarkably good job of keeping track of entropy from the different sources of randomness available in your computer, and it's extraordinarily unlikely that anybody would be able to figure out anything worthwhile about your PGP secret key by stealing and disassembling your computer weeks or months after you generated your key pair. But it's always nice to know that you have an even more secure option available. Some folks at UC Berkeley have compiled an excellent list of [web randomness resources](#), including source code and writings about PGP and /dev/random.

This circuit uses *avalanche noise* in a reverse-biased PN junction, the emitter-base junction of the first transistor. The second transistor amplifies it. The first two ALS04 inverters are biased into a linear region where they act like op amps, and they amplify it further. The third inverter amplifies it some more, and clips it to TTL levels. You can find out more about avalanche noise by reading the "Noise Diodes" section of the [Unusual Diode FAQ](#), or Terry Ritter's [notes on random electrical noise](#).



There are a couple of options for coming up with +18 volts. You can put two nine-volt batteries in series, or you can use a Maxim MAX232 line converter. The MAX232 part runs off five volts (which you'll need anyway for the 74ALS04) and has a clever switching circuit to generate +10 volts and -10 volts, which it uses for sending RS232 signals. So connect the +10 volts to the top of the 15K resistor, and the -10 volts to the emitter of the second NPN transistor (where I've shown a ground in the schematic). The output of the transistor stage is AC-coupled thru a capacitor, so this offset won't hurt anything, and there's nothing magical about 18 volts as opposed to 20 volts. Within reasonable limits, more is better, since you're trying to induce an avalanche effect in the emitter-base junction of the first transistor.

Now that you have the 74ALS04 and the MAX232 part on your board, you're all set to add a PIC controller to sample the output and format it into a stream of RS232 bytes. Just time bit lengths, add a start bit and a stop bit and a little time between consecutive characters, and ship them out. You can send them directly into the serial port of a PC, Linux box, or Mac. Macs

can accept RS232 signals using an RS422 cable, I believe.

So here is the sci.crypt article:

James Pate Williams has been establishing the good precedent of posting useful things to Usenet that we wouldn't want to lose (C source code for crypto-related algorithms). These are then picked up by Deja News and other organizations that regularly collect snapshots of Usenet and the web. In keeping with his precedent, and in view of recent events, I'm reposting a circuit from a couple of years ago which produces reasonably random bits at very low cost. With a little software help, these bits can be made arbitrarily secure.

Bits from this circuit have a lot of entropy, but they're not entirely unbiased. To generate 128 really random bits, collect 1000 or 2000 bits from this circuit, and hash them using MD5 or SHA. A hash ratio of 10:1 or 20:1 should be adequate to thwart any electromagnetic attack intended to influence or predetermine the final 128 bits. The bandwidth on this circuit is fairly high; the output of the final inverter can be sampled as often as every few microseconds. On an oscilloscope, obviously visible autocorrelations trailed off at about 200 or 300 nanoseconds. The higher the amplifier bandwidth, the lower the autocorrelation. It's a good idea to experiment with the R and C values and see if you can do better than I did.

One thing I've thought of doing but never quite had the energy was to make a board with one of these bit generators and a PIC controller. The PIC could format bytes for RS-232 transmission, and might even be capable of doing the hash. Many microcontroller evaluation boards include small areas of board space for little circuits, and this circuit should fit into at least some of them.

Reference to an earlier article from 1995, when I actually built and tested the circuit:

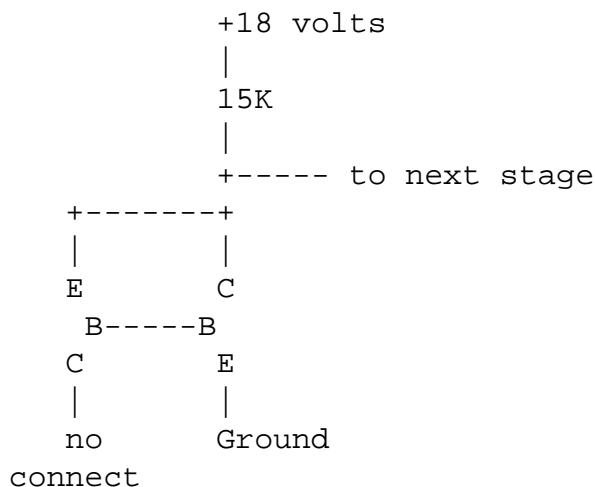
Subject: Re: How hard is it to make a RNG device?
 From: wware@world.std.com (Will Ware)
 Date: 1995/06/20
 Newsgroups: comp.security.misc,sci.crypt,alt.security

Paul Dokas (pbd@winternet.com) wrote:

: Roderick Smith (rsmith@psych.colorado.edu) wrote:
 : > To get a TRULY random number sequence, you'd need to rely upon some truly
 : > random phenomenon, like the decay of a radioactive isotope.
 : How hard would it be to make such a device? Aren't there analog electrical
 : devices that produce random noise that could be run through an A->D device
 : and then sent over an RS232 port?

This really should be in a FAQ somewhere. There are various ways to do this. People talk about using radioactive sources, and that probably gives better randomness, but a more practical approach is to use noise from a reverse-biased PN junction. Some people like Zener diodes for

this; I prefer vanilla NPN transistors, such as 2n3904s. I use two of them set up like this:



Then I use two stages of 74ALS04 inverters, biased into their linear range so they act like op amps. ALS gates have a good gain-bandwidth product and they don't ring (like AS gates do). AC-couple the input of the inverter with a 0.1uF cap, and put a 2.7K resistor from input to output as feedback/bias. Two stages like that, and connect the output of the second directly to the input of a third inverter. The output of the third inverter is a fairly clean TTL-level signal. It can be brought in one of the pins of a PC's printer port, or picked up in any other convenient fashion.

The signal should not be sampled more often than every few hundred nanoseconds, to avoid correlation between consecutive bits. There will be a noticeable bias (1s more likely than 0s, or vice versa). The bias can be greatly reduced by XORing a few raw bits together. Probably a better idea is to take a large number of bits and use a good hash function to reduce them to a smaller number.

I've built a few bit generators of this sort, and looked at their outputs on a spectrum analyzer. I didn't see any identifiable spikes. But if you wanted to be super-paranoid, you could encase the thing in a copper box surrounded by a mu-metal box, power it internally with two 9-volt batteries, and use the last inverter to drive an LED that shines thru a hole in the wall of the box to an external phototransistor (better use a real fast phototransistor).

But I suspect that the same level of security can be achieved far more easily just by hashing more bits together.

Nifty postscript!

Some students at Johns Hopkins have done some measurements on this circuit and found that its output looks good according to several statistical tests. (I've seen comments elsewhere expressing concern over the floating collector on the first transistor, and the possibility that the output is biased. To the first, I recommend tying the collector to the base. To the second, I agree, there's a bias, which is partly why I recommend hashing the output.)

I don't think I held onto a copy of their statistical results. I'll look around and include them if I find them.

Anyway, these folks at Johns Hopkins were interested in using the circuit as input to the /dev/random device in Linux. I contacted Theodore Ts'o, one of the original authors of the /dev/random code, and he explained how to do that:

```
From: "Theodore Y. Ts'o"
To: Will Ware
CC: gnu@toad.com, whgiii@openpgp.net, jsandin@chimera.acm.jhu.edu,
    wware@world.std.com
In-reply-to: Will Ware's message of Wed, 3 May 2000 11:48:38 -0400 (EDT),
    <200005031548.LAA05662@world.std.com>
Subject: Re: /dev/random questions
```

```
Date: Wed, 3 May 2000 11:48:38 -0400 (EDT)
From: Will Ware
```

Hi, a few years back I posted a web page describing an inexpensive hardware random bit generator using PN junction noise. I have recently been contacted by Joel Sandin at Johns Hopkins who is tinkering with the circuit. Wanting to be helpful (but being really pretty clueless about Linux kernel stuff), I started looking at the source for /dev/random, and noticed that the random_write() function adds new bytes to the entropy pool. I assume this gets invoked when somebody with root access does something like:

```
cat bunch_o_bytes > /dev/random
```

My question (beyond "is my understanding correct so far?") is, how do you tell /dev/random how many bits of entropy to expect for each bit you write to it? And how does one characterize the entropy of a source? Joel has been running some diehard tests and getting what look like good results.

In order to tell /dev/random how many bits of entropy, there is an ioctl, RNDADDENTROPY, and then a pointer to a structure which looks like this:

```
struct rand_pool_info {
    int      entropy_count;
    int      buf_size;
    __u32    buf[MAX_ENT_BUF_SIZE];
};
```

This ioctl requires root access.

The basic idea is that you have a user-mode daemon that receives the input from the hwrnd device driver, does whatever whitening, entropy characterization is necessary (which may potentially be CPU intensive, and so should be done in user-mode), and then uses this ioctl to feed the data into the /dev/random entropy pool.

I hope this helps! If you have any other questions, feel free to give me a yell.

The /dev/random source I've been looking at is at
<http://www.openpgp.net/random/>
I don't know if this really is current, it claims to be.
Thanks for any thoughts or suggestions.

I don't think it's the latest, but it's closest for government work.
Looking at the latest Linux sources is generally the best way of getting the very latest random code.

- Ted