T12 RANDOM NUMBER GENERATOR

T12RNG is a very low cost, simple and compact hardware random number generator (HRNG) that plugs into the ordinary PC serial port. It uses Atmel's ATTiny12 microcontroller to sample a whitenoise generator based on semiconductor avalanche noise.

Features

- Small, less than 3x3cm
- Plugs into the D-Sub9 serial connector common in most PCs
- Serial port powered, no need for batteries
- Very low cost, less than $10 in parts
- 60 random bytes per second average output rate
- Trivial protocol: hex digits output at 1200bps 8N1 can be seen with any terminal emulator

Applications

I originally developed T12RNG to create really unpredictable crypto key material, one-time pads/keys/passwords and such. This page describes the tests I ran to convince myself that it would be suitable for *my* purposes, but I don't claim to be an expert in crypto, random number generation or statistics. If you want to use this device for anything important, be sure to design and perform your own tests.

T12RNG's data rate -- 60 bytes/sec -- may be too slow for some applications, such as Monte Carlo integration. If that's what you're trying to do, perhaps you'd be better off with fast software-based pseudo-random number generators such as the Mersenne Twister.

You can use T12RNG to initialize or periodically shuffle the state of some other PRNG such as ISAAC to achive fast but highly unpredictable random sequences.

It may also be too fast -- 60 bytes/sec means almost 5MB/day. If all you need is just a few random bytes every now and then, discard all the rest.

Motivation

I'm working on transforming my aging iPaq H3650 in a kind of ARM-based crypto device for certain specialized applications. But since its storage is flash-based and it'll be used in a scenario with nearly zero external input, it is a prime candidate to the vulnerabilities described in Tzachy Reinman's paper and thesis.

Circuit

I wanted to make the circuit as simple and with as few components as possible, so I used the microcontroller's internal 1.2MHz RC oscillator as the clock source so as not to need an external crystal and their capacitors. I also made it transmit-only; there's no receiver, which means no configuration options nor in-system firmware upgradability.

Most of the circuit is simply a "copy and paste" from several ideas from other circuits I found googling around.

Two important tricks the circuit uses came from Murray Greenman's AVR-based DVM project: the method getting power from the serial port from the RTS and DTR lines using diodes and the idea of using the negative voltage 'mark' signal from the TX line to drive the RX signal when we want to transmit binary ones.

The whitenoise generator is from Aaron Logue's page, except that I changed the values of a few resistors to make the signal suitable for feeding directly to the microcontroller's analog comparator.

In order to generate a decent avalanche effect, we need at least 12V between the base-coupled Q3 and Q4 transistors; to achieve this, I feed them TX and RX. Given that TX is normally below -6V and RX is normally above 6V on most serial ports, there should be at least 12V between them. The rest is just amplifying the signal with Q5 and DC-ofsetting it with the two resistors to make it center around 1.1V, which is the microcontroller's on-chip internal reference voltage fed to the analog comparator. So, the analog comparator output should be a random bit dancing to the sound of the white noise.

Your operating system software must raise the DTR and RTS lines when opening the serial port, otherwise the circuit will not turn on. Lowering those lines turns the circuit off, saving a few milliamps. This may may be useful in low power scenarios. Furthermore, your serial port should provide at least +/-6V, which means that this device may not work with a few ultra-low power serial ports found in certain notebooks that deliver only meager +/-3V. Most PC serial ports deliver well over +/-9V, however.

The circuit in the picture does not have the In-System Programming (ISP) header needed for uploading the firmware to the microcontroller. It is assumed you have a programming dongle or evaluation board to do that. The ZIP package has a version of the circuit that does have this header. The device on the left of the picture is the prototype with such a header. I took it out on the final PCB version to make the device smaller. It can be made even smaller by using surface mount components.

Firmware

The firmware samples the analog comparator output 9600 times per second. If the source is truly random and the signal averages at exactly the reference voltage, this should theoretically give us pure random bits with in an exact 50-50% proportion of zeros and ones.

But because of component tolerances, temperature variations, material imperfections and many other factors, this proportion is more likely to be biased one way or the other.

Because of this, the random bits are passed through a 'debiasing' algorithm due to Von Neumann: they're grouped in pairs and if they're both equal, we discard them. Otherwise, the last one is fed to a shift register. This is repeated until we get 8 bits, forming a random byte.

Any good information theory or crypto book explains why process this removes the bias, straightening the propotion to nearly 50-50% regardless of what the original bias was. This also has the effect of dividing the output rate by four on average, so we should be getting 1200 random debiased bits per second or about 150 bytes per second. It can get much slower than that though if the original bias is too strong, say 75-25% or 90-10%. Even with decently small bias, it is possible that we get an unusually long sequence of only zeros or only ones, although the chance of this happening by chance gets small very fast as the lenght of the sequence increases.

This random byte is then mixed with the current item in a 10-byte circular buffer and the current item is incremented. In earlier versions, the mixing function was just a XOR. However, this made the device generate very nice random numbers when connected to some computers but not others. While searching for a cause, I decided to experiment with a more complicated mixing function with additions and taking inputs from the timer. While it's a bit of hocus-pocus not very well backed by theory, the results then passed all statistical tests in every computer I attached the device to. Go figure.

The transmission process runs simultaneously with the sampling process. One byte is taken from the pool, converted to two hex digits and transmitted a by 'software UART', since the ATTiny12 microcontroller does not have a built-in hardware UART. This software UART simply transmits the start bit, data bits and stop bits at the appropriate times, at 1/8th of the sample rate. In other words, the data is transmitted at 1200 bps, 8 bits, no parity, one stop bit. This yields 120 hex digits per second or 60 random bytes per second.

Every 78 hex digits have been transmited, a line break (CR-LF or "\r\n") is sent just to make output look nice in the terminal emulator.

Since the transmission process happens simultaneously with the sampling/debiasing/mixing process, it is important that we generate random bits faster than we transmit them. We transmit at constant 60 bytes per second, but while the generation speed is often faster (it is often the case that the 'head' pointer used in insertion runs past the 'tail' pointer used for getting bytes for transmission), it is not *guaranteed* to be faster.

For instance, the avalanche effect is known to decrease with temperature. This could reduce the generation rate up to the point that it becomes consistently slower than the transmission rate. Because of that, I implemented an additional precaution: there is another 10-nibble (5-

byte) buffer that counts how many times a particular item in the pool was exclusive-or'ed. We refuse to transmit a byte if its corresponding counter is zero; in this case, we transmit two 'dashes' to give time for the generator to recover and feed more randomness to the pool. On the other hand, if the counter is non-zero, we transmit the byte as two hex digits and clear its corresponding counter. In other words, we keep track of whether the generator is behind and transmit dummy data when it needs time to recover.

This precaution also prevents that catastrophic failures would happen silently. Say that after thousands of hours of correct operation, one of the 2N3904 transistors in the generator blows up (this is unlikely but possible). Or that you inserted it with enough force to make a leg in one of the transistors pop its solder away (I actually did this once -- that's why I filled that section of the circuit with silicone glue). That would cause the device to keep repeating the last 10 bytes over and over because the transmitter would keep running while the pool would stop being replenished, since the constant stream of zeroes would cause the debiaser to discard all of them.

With our precaution, the generator would simply start transmitting endless dashes. The upper layer software could then raise an alarm if it receives nothing but dashes for more than a certain period, say, one or two seconds. Receiving a few dashes every now an then, say, less than eight, is normal -- that's when the generator gets behind the transmitter.

When the device starts, the pool is empty and it would transmit dashes until the generator leaps ahead of the transmitter. We avoid this small ugliness by transmisting a brief version and copyright message.

Testing

For the initial tests, I thought of using John Walker's `ent` program. It computes a few statistics that can be used to determine whether the data looks random or not, such as average, entropy, chi-square sums, serial correlation, etc. See his page for a nice explanation of what all that means.

But `ent` operates on files; impatient that I am, I didn't want to spend several hours waiting for enough random numbers to accumulate to see a (possibly bad) result. I wanted to see how it performs *over time*, taking a look every now and then to the statistics so far while I was doing other things.

So I wrote a Perl script based on `ent` that computes the statistics *as the data is read from the serial port*, showing partial results once a second. I changed added a few other stats (data rate in bytes per second, proportion of zeros to ones) and changed the presentation of a few others: instead of showing the values themselves, I show the deviation from the ideal values. I didn't include the Monte-Carlo Pi calculation, since it converges too slowly.

I rewrote most of this program in C to perform another function besides computing the scores: it accumulates 10kbytes of random data from the generator and tests it. If it passes certain quality criteria, it is sent to a queue of approved bytes. This queue is then sent to the Linux Entropy Pool `/dev/random` device. Ok, just doing that doesn't really solve all the vulnerabilities, but may be good enough for certain applications.

Source :
http://www.postcogito.org/Kiko/T12RandomNumberGenerator.html#Testing